Netherlands organization for applied scientifi research

TNO-report

AD-A245 620

TNO Physics and Electronics Laboratory

P.O. Box 96864
2509 JG The Hague
Oude Waalsdorperweg 63
The Hague, The Netherlands

Fax +31 70 328 09 61
Phone +31 70 326 42 21

1

report no.	copy no.
FEL-90-A012	8

title

QUEST: Quality of Expert Systems

author(s):

Jacques H.J. Lenting (RL)
Michael Perre (TNO-FEL)

date:

November 1990

classification

title	: unclassified
abstract	: unclassified
report text	: unclassified
appendices	: -

no. of copies	: 40
no. of pages	: 91
appendices	: -

TNO

Accesion For		
NTIS CRA&I	✓	
DTIC TAB	☐	
Unannounced	☐	
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and/or Special	
A-1		

report no. : FEL-90-A012

title : QUEST: Quality of Expert Systems

author(s) : Jacques H.J. Lenting and Michael Perre

institute : TNO Physics and Electronics Laboratory

date : November 1990

NDRO no. : A89K602

no. in pow '89 : 704.2

ABSTRACT (UNCLASSIFIED)

This study contains a summary of the results of the technology project "QUEST: Quality of Expert Systems", carried out under commission of the Dutch Ministry of Defence, Directorate Defense Research and Development. Participants in the project are TNO Physics and Electronics Laboratory (TNO-FEL), University of Limburg (RL) and the Research Institute for Knowledge Systems (RIKS). After an analysis of the problems encountered in expert systems development, a quality framework is developed which takes a view at the quality problem from three perspectives: the quality of the development process, the quality of the specifications and the quality of the expert system viewed as a product. In order to get a better grasp of the problem a number of methods and techniques, derived from conventional and artificial intelligence systems development, are reviewed. Secondly, the conceptual similarities between databases and knowledgebases are stressed. The use of conventional specification methods, in particular Nijssens Information Analysis Methodology (NIAM), is considered. In addition to this algorithms for preserving consistency and integrity of the knowledgebase are compared. Thirdly the modularity and structure of knowledgebases is examined, together with the applicability of conventional testing methodologies in expert systems. Lastly it is demonstrated that the integration of database theory and artificial intelligence signifies a step in the direction of a better quality control of expert systems.

rapport no.	: FEL-90-A012
titel	: QUEST: Kwaliteit van Expertsystemen
auteur(s)	: Drs. J.H.J. Lenting en Drs. M. Perre
instituut	: Fysisch en Elektronisch Laboratorium TNO
datum	: november 1990
hdo-opdr.no.	: A89K602
no. in iwp '89	: 704.2

SAMENVATTING (ONGERUBRICEERD)

Dit rapport bevat een samenvatting van de resultaten van het technologieproject "QUEST: Kwaliteit van Expertsystemen", uitgevoerd in opdracht van het Nederlandse Ministerie van Defensie, Directoraat-Generaal Wetenschappelijk Onderzoek en Ontwikkeling. Deelnemers aan het project zijn het Fysisch en Elektronisch Laboratorium TNO (FEL-TNO), de Rijksuniversiteit Limburg (RL) en het Research Instituut voor Kennis-Systemen (RIKS). Na een analyse van de problemen die zich voordoen bij de ontwikkeling van expertsystemen wordt een raamwerk ontwikkeld dat het kwaliteits-probleem benaderd uit drie gezichtspunten: de kwaliteit van het ontwikkelproces, de kwaliteit van de specificaties en de kwaliteit van het expertsysteem als produkt. Om een beter begrip van het probleem te krijgen worden methoden en technieken besproken om de kwaliteit van systemen te beheersen. Daarbij wordt zowel verwezen naar conventionele als kunstmatig intelligente systeemontwikkeling. Ten tweede is er aandacht voor de conceptuele overeenkomsten tussen gegevensbanken en kennisbanken. Implicatie hiervan is het gebruik van Nijssens Informatie-Analyse Methode (NIAM) als specificatiemethode voor kennisbanken. Verschillende algoritmen gericht op het waarborgen van consistentie en volledigheid van de knowledgebase worden met elkaar vergeleken. Ten derde is het onderwerp modulariteit en structurering van kennisbanken onderzocht met daarnaast aandacht voor de toepasbaarheid van conventionele testmethodologieën op expertsysteem-software. Ten slotte wordt aangetoond dat met de integratie van gegevensbanktheorie en kunstmatige intelligentie een belangrijke stap kan worden gezet op het pad naar een betere kwaliteitsbeheersing van kennissystemen.

1 INTRODUCTION

Since the fifties, research in the field of artificial intelligence (AI) aims at the development of systems that manifest themselves in new functions such as visual perception, natural language processing, learning and expert systems. The objective is to capture the input-output behaviour of the system with regard to these specific tasks, that can match its performance with that of a human being. This comes down to modelling cognitive processes in such a way that these can be performed by a computer system. This activity resembles the modelling of other (technical) systems, that are familiar to science. The specifications of a system, for instance, determine the level of detail and (un)certainty required by the model, both for static and dynamic behaviour.

However, there is an essential difference between the accepted concepts, methods and tools. The construction of qualitative models lies at the centre, i.e. models that establish explicit specifications of the (problem) domain and the solutions/heuristics in terms of *facts* and their relevant interrelations, rather than numerical descriptions. To execute these models, the information processing capabilities of the computer are extended with a derivation component, usually based on logic.

It is expected that conventional, automated systems will in future contain AI components. Reliability amongst other things, must be assured if these systems are to be used for critical tasks. It is a well-known fact that the development of large systems is plagued by exceeding the project funds and delivery dates. Then, when the system obtains operational status, it often turns out to be functionally defective. Quality research receives much attention in the fields of both conventional and AI systems. Very strict demands are put on military and critical civil applications, especially those concerning reliability and fail-safe operations. Examples of military systems are, for instance, Command, Control, Communication and Intelligence systems (C^3I); critical civil applications are process control systems for chemical and power plants.

During research into the quality of automated systems, one is confronted with a number of problems. Firstly, there is no generally accepted definition of the notion of quality. The factors that make up the quality of a system are important entities to determine with which quality can be measured. Secondly, a system's quality is measured after it has become operational. Defects apparent in this phase, can only be solved with large investments, both in time and money. To ensure a proper quality assurance of automated systems, it is necessary to take this aspect into account during the very first phases of system development. The emergence of expert systems as a commercial product stems mainly from the principle of software reuse, e.g. in expert system shells. Other types of software too, such as relational database management systems (RDBMSs) have benefited from this principle. In this study, therefore, expert system development will primarily be viewed as knowledgebase development.

This study attempts to present a survey of methods and techniques available to control the quality of expert systems. Three aspects related to the quality-problem can be distinguished, i.e.: the development process, the specification and the product. This study focuses at that part of system development that sets expert systems substantially apart from other software: the specification and implementation of the knowledgebase. The methods and techniques selected for the development of expert systems are based on, or are directly derived from (relational) database technology. Though it is has not yet been established that the relational database approach is adequate for all types of expert systems, it seems that this approach offers a better perspective where quality assurance is concerned, than any other development methodology. This study delineates a quality framework in terms of process, specification and product, on the understanding that 'the process' refers to the comprehensive process of expert system development (the life cycle), whereas 'the product' and 'the specification' relate to parts thereof, i.e. the knowledgebase and the knowledge model.

Before entering the subjects of the next chapters, it is important to know what exactly makes up an expert system. Expert systems form a new class of automated systems that have enjoyed a large interest over the past few years. They form one of the practical results of the research into the field of artificial intelligence.

While building expert systems, methods and techniques are applied to create man-machine systems that can solve problems in a specific area. To do this, knowledge, experience (or expertise for that matter) of a human expert is needed. Beside theoretical knowledge from textbooks, this may also exist of more general experience rules or rules of thumb. Another name for this quite vague knowledge is heuristic. 'Heuristics enable the human expert to make, where necessary, educated guesses, to recognise promising approaches to problems and work effectively with incomplete or even wrong data' [Hayes-Roth83].

Knowledge lies at the centre of expert systems. A number of reasons can be given for this:

- certain problems can hardly be solved or cannot be solved at all with the help of fixed algorithms;
- a person qualifies as an expert because of the very fact that he possesses more than only theoretical textbook knowledge, his heuristics so to say;
- knowledge is a scarce commodity that can be communicated to a larger group of persons by means of an expert system.

Expert systems are based on symbolic processing, rather than conventional systems that emphasize numerical processing. Pertinent differences are shown in figure 1.1.

	Processing method	
	Symbolic	Numerical
Technique used	heuristic search	algorithmic
Definition of solution steps	not explicit	explicit
Answers	satisfactory	optimal
Procedures / data	separated	interwoven
Knowledge	not exact	exact
Adaption	often	seldom

Figure 1.1: Differences between symbolic and numerical processing.

Expert systems usually consist of four elements; the knowledgebase, a reasoning mechanism, an explanation facility and a man-machine interface. The knowledgebase holds facts and rules that represent the actual knowledge of the system. The reasoning mechanism reflects the way of reasoning used in a specific problem area. It consists of procedures with which new knowledge can be derived on the basis of available knowledge (or information). Expert systems offer the user advice with regard to solving certain problems. The user's confidence in this advice is strengthened if the system is able to explain how the actual result has come into existence. Furthermore, the system must sometimes query the user, especially when it is not able to elicit certain data from the knowledge available. At these moments, a user wants to know why the question arises. From the foregoing, it appears that the user interface is vital to the system's functioning. The man-machine interface will have to take care of this.

Expert systems are used for various problem categories. The most apparent is fault diagnosis. The word fault must not be taken too literally; especially in the field of medical diagnosis, expert systems have come into their own successfully. The MYCIN system [Shortliffe76], applied for the diagnosis in the field of internal medicine can be seen as a milestone in the development. Other applications are the interpretation of data to describe a certain situation realistically, as for example, signal interpretation systems (sonar and radar). Monitoring systems are of a somewhat more difficult type. These systems are meant to keep an eye on a process. This involves measuring process parameters; should the values measured deviate from a predefined norm, the system will diagnose the problem after which possible counteractions will be planned. However, planning of activities is a difficult task for an artificial intelligence system. Applications of this type are therefore scarce.

It remains hazardous to sharply define expert systems. It may be compared with shooting at a moving target. The moment a system can execute a certain task, the intelligence needed for execution is sometimes forgotten.

Especially [Nijssen86] expands on the thesis that expert systems are nothing but automated information systems containing a certain amount of human expertise. This expertise (or knowledge) is merely a set of interrelated facts. Facts that could be entered by the user himself (as the result of a query or during the creation of the database). It is also possible that they were derived from facts already present in the database with the help of derivation rules. Lastly, they can be the result of a reasoning process: by applying inference rules to the facts already present, new facts can be inferred. Figure 1.2 shows the structure of what can be termed a relational expert system. It demonstrates the equivalence of databases and knowledgebases.

A relational expert system has a structural section containing the definitions of the tables, a section with validation rules, a section with derivation rules (that can infer facts that are implicitly contained in the database) and a section with inference rules (that can infer facts that are not implicitly contained in the database). Users converse with the system and operate their own database, next to that there is a virtual database containing derived and inferred facts.

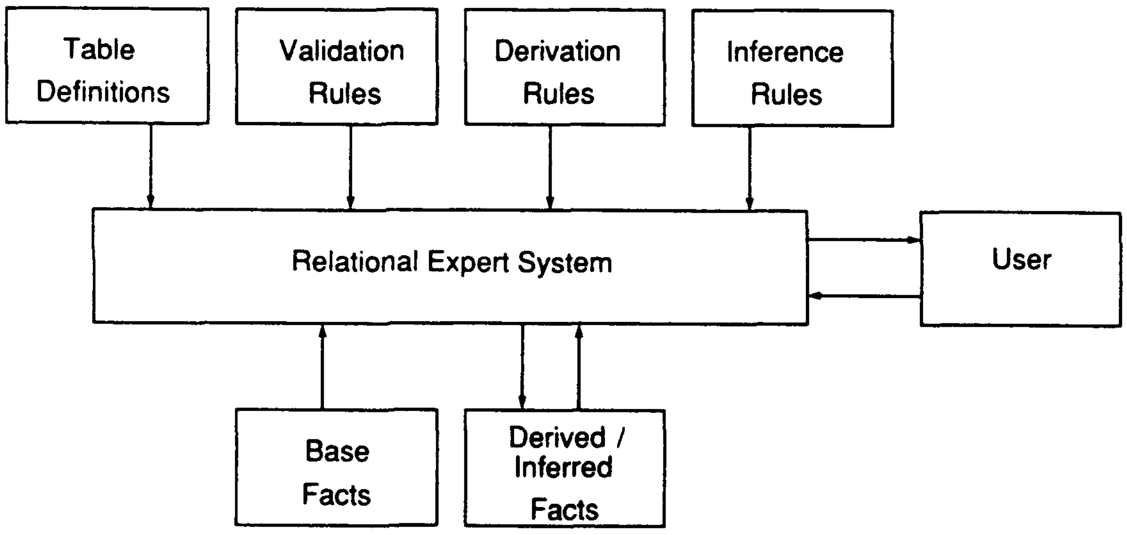


Figure 1.2: A relational expert system.

Concepts such as data independence, data integrity, controlled redundancy, security and privacy that are used in an RDBMS are also important to expert systems. Other reasons to hold on to the relational model are its conceptual simplicity and the fact that system developers are familiar with it. By using an RDBMS, update anomalies can be prevented, just like redundant storage. Moreover, it offers a flexible growth potential when operational concepts change and alterations must be made to the data structure.

By considering a knowledgebase as a special kind of database, various facilities available in DBMSs, can also be used in knowledgebase management systems (KBMSs). Examples are recovery (the repair of a database after a calamity, an error or power failure), concurrence (the simultaneous use of a database by different persons), security (the protection of a database against unauthorized use) and integrity (guarding the consistency of the database). Especially with regard to the latter point, a direct relationship can be made with analysis and design methods for databases. Globally, this implies that a conceptual model of the knowledge domain must be made. The conceptual model lies somewhere in between the internal model of the database (the way in which relationships are physically incorporated into the database) and the external model (the way in which the user views the system). The conceptual model must be a complete and consistent representation of the knowledge domain, where a distinction is made, just as in a database application, between a knowledge scheme (definitions of all facts and relations present) and the actual knowledgebase. This distinction can also be taken as a clear separation between types and instances. The advantage is that the knowledgebase can now be discussed on an abstract level. Consistency and completeness can be monitored by using constraints that are applied to the actual knowledgebase.

The reports on which this outline is based are [FEL1989-148], [FEL-89-A267] and [FEL-90-A312].

The remaining chapters of this study deal with the following subjects. Chapter 2 contains a further analysis of the quality problem in expert systems. On the basis of this a quality framework is developed, departing from three approaches: i.e. process, specification and product with their related quality criteria. Chapters 3, 4 and 5 elaborate on these approaches. Chapter 6 deals with an aspect of expert systems that is essential to quality assurance, i.e. the integrity of the knowledgebase. Finally, chapter 7 contains the conclusions.

2 A QUALITY FRAMEWORK

2.1 Introduction

The quality of expert systems is generally regarded as disappointing, certainly when compared to more conventional software. In particular knowledge acquisition, testing, evaluation and maintenance of knowledgebases are deemed to be problematical aspects. As yet, there is little agreement when it comes to the way in which these problems should be dealt with. An important motivation for research into the quality of expert systems is the fact that conventional systems make increasing use of intelligent modules, without complying to the same, strict quality requirements that are put on the conventional part.

A number of aspects influence the requirements related to quality. Type, application area and scope of the system come to mind. If the system is meant to serve as a research prototype, then it will need less stringent quality requirements than an operational system does. The role of the application area is also important, especially in the case of critical applications, such as process control systems in chemical industry or power plants. The same goes for military command systems, usually termed C^3I systems (Command, Control, Communications and Intelligence) in military jargon [Napheys89]. Such a system needs more stringent quality requirements than an administrative one. Automated systems sometimes are very large and therefore difficult to oversee. Information analysts must communicate system specifications to programmers. Using a quality standard assists in controlling such a project.

This chapter first presents an analysis of the problems that are encountered in controlling the quality of expert systems. Next, a quality framework will be developed, with a number of quality criteria related to it.

2.2 Problem analysis

Quality assurance performed along the lines of the framework presented for this purpose can contribute to the development of better systems. Quality assurance implies the realization of a good product (eventually to be measured through user appreciation) as well as meeting the ts established time-related and financial conditions. The quality assurance of expert systems leaves much to be desired, as is apparent from a multitude of complaints, such as:

- frequently exceeding the budget;
- disappointing final product, e.g. in the area of:
    - connectivity with other (conventional) software;
    - flexible user interaction;
- lack of maintainability of the product.

Such a list of complaints is not substantial enough to evaluate the seriousness of the problems, let alone to determine countermeasures. To investigate in how far these complaints are specific or generic for expert systems, it is necessary to relate them to the problems that lie at the root of them. Making an inventory of the activities of the development process where most problems occur, is the first step. After that, the next can be identified:

- specification;
- knowledge acquisition (=knowledge acquisition + modelling);
- evaluation;
- testing;
- maintenance.

The all-embracing problem in the development of many expert systems is the absence of clear, hierarchically refined specifications. This is especially true for so-called expert emulation systems. The initial issue, the emulation of some kind of expert, does not lend itself very well to systematical refinement. This is probably partly due to the fact that methods and techniques of the AI discipline are not yet fully developed. It is doubtful whether conventional techniques can offer a satisfactory answer. Automating (or effectively supporting) a medical diagnosis is more difficult than automating a corporation's administration because of the considerably lower level of domain formalization involved.

Knowledge acquisition is generally considered to be the bottle-neck of expert system development [Breuker88a]. Many expert systems developed in the past have made it clear that eliciting knowledge from a human expert and making it a coherent entity, is a laborious and time-consuming process. The main reasons are the fact that:

- knowledge of the human expert is mostly compiled (difficult to make explicit);
- there is no absolute consensus among human experts.

Evaluation deals with the functionality of the system. In many expert systems, the evaluation of the accuracy in particular, is a problem. This evaluation problem occurs when concrete validation of the system's output is not possible, because the correct answer is unknown or because there are more (reasonably) correct answers. Beside the direct consequences of this tiresome evaluation for quality assurance, there also is an indirect effect. Namely, the extra requirements with regard to the functionality of the system, for instance, in the form of flexible user interaction and an explanation that is readily understood by the user.

Testing capacity is related to the evaluation capacity. However, evaluation is a matter (or at least, could be) of judging the final product, whereas the system's testing capacity is inseparably connected to a hierarchic specification with a corresponding, systematic subdivision of software into smaller, distinct subsystems [Myers78]. In most of the expert systems developed so far, such a hierarchical structure is hardly to be found. Observations like 'Expert systems often have a convoluted, intricate flow of control' [Llinas87] also seem to point in this direction. Beside the lack of hierarchical structure, the use of heuristics and incalculable entities too, frustrate the testing of expert systems.

The problem of maintenance also leads back to what principally is the lack of hierarchy as well as modularity in the average knowledgebase. This lack is especially evident for those domains where knowledge has become the subject of frequent change.

Together with the issue of maintenance, the defective explanation facilities of expert systems too, are seen as an obstacle [Clancey81,Breuker88a]. Should one succeed in hierarchically structuring a knowledgebase in a useful way, then the explanation could also be improved. Proper explanation facilities are especially important in expert systems that suffer from an evaluation problem. If no satisfactory, objective evaluation entity can be found, a good explanation is indispensable.

The expert system software crisis is not yet solved if a solution would be found to these subproblems; a coherent, generally applicable methodology is called for, as is a uniform and generally applicable specification language. These, however, are aims that can only be realized completely when these subproblems have been solved. Furthermore, it is doubtful whether an appropriate, uniform methodology can be found.

Because in practice, the label expert system is put on significantly different programs and every definition encountered in the respective literature leaves room for further interpretation, the validity of many claims about expert systems in general becomes quite debatable. Knowledge acquisition as a bottle-neck in the development of expert systems, for instance, has recently been contested by [Cullen88]. Claims about quality assurance are no exception in this respect. To prevent claims that were legitimately made, for example, about MYCIN (or with MYCIN in mind), from being generalized without second thought into the realm of expert systems in general, the recognized problems must be coupled with the characteristics that are necessary or sufficient to make these problems appear.

Figure 2.1 takes a first step in this respect. The diversity of expert systems is made explicit in the form of ten dimensions in which individual expert systems can differ and that are important with respect to the amount of problems with which one will be confronted during the development of a specific expert system.

These dimensions are subdivided into three types: domain-dimensions, structure-dimensions and functional dimensions. The first two relate to the field of knowledge and what may be expected from the expert system in absolute terms. Characterizations of these dimensions must therefore be classified as preconditions. As far as functional dimensions are concerned, domain and global specifications are, within certain limits, controlling factors.

Structure-dimensions relate to the structure of knowledgebases, the knowledge representation so to speak. The characterization of an expert system in terms of these dimensions is mainly a matter of choice for the designers.

A number of these dimensions deserve further discussion. Conceptual complexity, the number of object and relation types (predicates) is a consideration. Knowledge extensiveness refers to the (average) number of specific objects or relations per type, in other words, the average number of instances.

Hierarchy and modularity are seen as separate dimensions because systematic subdivision of a knowledgebase (hierarchy) does not guarantee that changes in one of the component parts will not yield unexpected effects concerning the functionality of the other parts (modularity).

Dimension	Type	Danger zone	Associated problems
Controllability of the final result	domain	low	evaluation explanation
Accessability of knowledge	domain	low	knowledge acquisition explanation
Consensus about knowledge	domain	low	knowledge acquisition evaluation explanation
Conceptual complexity	domain	high	knowledge acquisition
Knowledge extensiveness	domain	high	knowledge acquisition testing
Hierarchy	structure	low	testing maintenance explanation knowledge acquisition
Modularity	structure	low	testing maintenance
Uncertain inference	functionality	high	knowledge acquisition testing
Expert emulation	functionality	high	knowledge acquisition explanation
Creativity	functionality	high	knowledge acquisition

Figure 2.1: Dimensions of an expert system in relation to quality assurance.

Uncertain inference implies the use of certainty factors or other confidence factors as well as the use of heuristic rules.

Creative systems refer to design systems that are not based on decision trees, in other words, that find other ways to a solution than making a selection from a limited number of alternatives.

2.3 A quality framework

Unnecessary to say that claims about the quality of software are only possible when the notion itself has been properly defined and can be measured. Whether a piece of software has a good quality is difficult to establish; the absence of quality, however, is much easier to demonstrate. This section intends to build a framework in which the quality assurance of expert systems can take place. Three aspects are important:

- analysing a (object) system which then will result in system specifications;
- development process of expert systems;
- expert system as a product.

The relationship between these three aspects can be represented as follows:

product = f{development process (system specifications)}.

Applying the development process to the specifications results in the expert system as a product. Quality assurance thus takes place in three ways:

- validation and verification of system specifications;
- structured development process;
- testing and evaluating the product.

Because the system specifications are of great importance during the whole development phase of a system and many errors are only detected after implementation, in this study special attention will be focused on quality assurance. During validation of system specifications it is determined whether these are in accordance with the user needs. During the validation process, the specification is confronted with reality. During verification the specification is confronted with the implementation.

The need for a structured development process has long since been recognized. Phasing methodologies such as System Development Methodology (SDM) play an important role in this respect [Turner87]. In the field of expert systems development, a methodology is used that was derived from SDM, namely Structured Knowledge Engineering (SKE) [SKE88]. For specific activities in expert system development, e.g. knowledge acquisition, this methodology provides detailed guidelines. During the various stages of development, (sub)systems must be tested with regard to functionality and user needs. Testing can be considered as observing the behaviour of a (sub)system when it is fed with a controlled selection of data. During evaluation, experiments are carried out after which the results are compared with the solutions for a specific problem given by a recognized expert.

2.4 Quality criteria

Llinas and Rizzi [Llinas87] present the following list with test and evaluation criteria:

- consistency and Completeness of the knowledgebase;
- software quality criteria;
- man-machine interface;
- costs;
- quality of explanation;
- relevance;
- reliability;
- portability;
- contribution to organizational aims;
- quality and accuracy of decisions, advice or recommendations;
- correctness of reasoning;
- system efficiency;
- adaptability (maintainability);
- run-time;
- computational efficiency;
- intelligent behaviour.

This list, however, is not a good starting point, because it contains too many vague and overlapping items. Moreover, some of the criteria (consistency and completeness, run-time) fall within the scope of units of measurement, whereas others (software quality criteria, costs, contribution to organizational aims) provide a very trivial elaboration of the philosophies associated with them.

Subject of this study is the quality assurance and especially the methods and techniques that can lead to a product of better quality. Figures 2.2, 2.3 and 2.4 present a vision on quality. Quality aspects of specification, process and product are given separately, for easy interpretation of the used terms. Nevertheless some terms deserve a bit of explanation. The terms consistency and completeness are used in the literature for quite different notions, depending on context.

```
|  usefulness
|  |     reliability
|  |     |     robustness (x)
|  |     |     controllability (x)
|  |     efficiency
|  |     |     task division (x)
|  |     |     task allocation
|  |     |     |     capability
|  |     |     |     portability (x)
```

Figure 2.2: Quality criteria for the development process.

Falsifiability of specifications refers to the fact that specifications that are vague to such an extent that even very diverse implementations can be verified with them, are rather useless. In the case of robustness of the developing process, one must think of escape clauses that anticipate unforeseen set-backs and absorb these as good as possible. The robustness of a product refers to the protection against unexpected external influences. These can be of a human or technical nature. Fail-safe refers to responding to technical defects such as power failure, whereas monkey-proof applies to human misbehaviour behind the keyboard.

Especially expert systems evoke a call for the specification at knowledge level [Newell82,Breuker87] at a large distance from the implementation. This in itself may be very advantageous, provided that this distance can be bridged in a formalized manner through a (number of) specification(s), which are much closer to the implementation.

```
|   usefulness
|   |     reliability
|   |     |     integrity
|   |     |     |     consistency (x)
|   |     |     |     completeness (x)
|   |     |     testability
|   |     |     |     modularity (x)
|   |     |     |     falsifiability (x)
|   |     |     efficiency
|   |     |     |     ease of design
|   |     |     |     ease of development
|   adaptability
|   |     maintainability
|   |     |     readability
|   |     |     modularity
|   |     portability
|   |     |     hardware independent
|   |     |     implementation independent
```

Figure 2.3: Quality criteria for the specification.

The layout of the diagrams indicates the hierarchy of the concepts. Usefulness and adaptability, for instance, are seen as the most important quality aspects of specifications, with usefulness being subdivided into reliability and efficiency. For a better understanding, one must realize that some (actually most) of the criteria in the diagram and entities are desiderata of the specification, the process or the product, whereas others are desiderata of the techniques that are to be applied. The latter category, for instance, includes ease of design and development in specifications. Quality aspects that will receive most attention in this study are marked with an (x). A more detailed explanation of the selection of just these aspects as key notions of the quality of specification and product, will be given in the respective chapters.

```
|  usefulness
|  |  reliability
|  |  |  integrity
|  |  |  |  consistency (x)
|  |  |  |  completeness (x)
|  |  |  correctness
|  |  |  robustness
|  |  |  |  fail-safe and monkey-proof
|  |  |  |  graceful degradation
|  |  |  testability
|  |  |  |  modularity (x)
|  |  |  |  closely related to specification(x)
|  |  efficiency
|  |  |  computational efficiency
|  |  |  user friendliness
|  adaptability
|  |  maintainability
|  |  |  readability
|  |  |  modularity (x)
|  portability
|  |  hardware independent
|  integratability (connectivity to other software)
```

Figure 2.4: **Quality criteria for the product.**

3 THE LIFE-CYCLE OF AN EXPERT SYSTEM

3.1 Introduction

Practice reveals that the development problems of many expert systems can essentially be reduced to management problems [Cullen88]. This seems to be trivial and certainly is not specific to the development of expert systems, yet during development this tends to be forgotten. As such, in practice it may turn out to be the largest bottle-neck. Serious quality assurance of an expert system must insist on management support of the product under development, and phasing must include enough room for a feasibility study and the establishment of an evaluation framework in an early stage. This chapter deals with a number of project phasing methodologies that can act as guidelines during the development of an expert system.

3.2 A conventional look at an unconventional development

Most development methodologies for expert systems emphasize the knowledge acquisition phase. This is no wonder, when we consider the degree of complexity, intrinsic to this activity. However, the development of an expert system is not limited to this phase only. Especially the Structured Knowledge Engineering methodology (SKE) [SKE88] makes a (concise) statement, based on the System Development Methodology (SDM) [Turner87], about the arrangement of the other phases, such as technical design, implementation, testing, acceptance and maintenance. The advantage of the SKE approach is that it embraces a familiar, conventional method of development improves the chance of actual application.

SDM is a phasing methodology for the management of system development projects. SDM deals with the whole of a system's life-cycle, from information planning through to maintenance. The SDM manual is a guideline for managing projects, arranging the documentation, development and management of information systems. For every development phase, the activities to be performed are included. For every activity are registered: a description, one or more analysis and design methods (e.g. NIAM, JSD or SA), a number of deliverables or documents and literature references.

This manual fulfils the role of a cookbook; not all of the details are important to every project. By considering it as an extensive check-list, it is possible to make plans, guide development team members through their tasks and establish standards for documentation. Special attention is paid to human interaction with the system, which benefits the development of an adequate man-machine interface.

SDM consists of the following phases:

- information planning: the development of an information plan in accordance with the aims of the organization;
- definition study: defining the system specifications, choosing a method to develop the system and making an initial cost-benefit analysis;
- basic design: refining the system specifications and a first system design;
- detailed design: designing subsystems and their components;
- implementation: acquisition of devices and software, after which the system will be programmed and tested;
- installation and conversion: transition to, and utilization of a new system;
- use and management: maintaining the system in such a way that it meets the specifications that evolve from the organisation.

The acceptance and integration phases will not really differ from their conventional system development counterparts. This does in no way imply that these phases would be of lesser importance. Emphasis must go to the phases regarding knowledge and information requirement analysis up to the implementation of the expert system.

The first, more or less structured development cycle for expert systems has been described by Shortliffe and Davis [Shortliffe75]. This cycle is known under the name of SIGART development cycle because it was mentioned for the first time in the SIGART Newsletter no. 55.

The SIGART development cycle has the following phases:

1. top-level design: definition of long-term objectives;
2. implementation of a Mark I prototype to demonstrate its feasibility;

3. refinement of the system, usually by:
    a. running informal test cases to elicit feedback from the expcrt, which results in a refined Mark II prototype;
    b. making Mark II available to 'friendly' users;
    c. revising the system on the basis of user feedback;
    d. making the revised Mark II available to users and return to phase 3b;
4. structured evaluation of performance;
5. structured evaluation of user acceptability;
6. operational use in a prototype environment for an extended period;
7. making follow-up studies to investigate whether the system is ready for large scale operation;
8. adapting the program so that it is suitable for a wide distribution;
9. general release and marketing with clearly-defined plans for maintenance and updating.

The SIGART life-cycle further consists of numerous recommendations, but does not contain any further methods.

Llinas and Rizzi [Llinas87] state that this phasing is essentially a rapid prototyping approach. They even go further: they believe that the development process of (non-trivial) expert systems is, by necessity, of an evolutionary character, as long as there is no better understanding of the knowledge engineering process. They draw a parallel with the early days of the development of FORTRAN, before the days of programming paradigms when program development was performed in exactly the same way as in the SIGART development cycle.

3.3 Rapid prototyping

As we have already mentioned in the previous section, rapid prototyping is a very important paradigm in the development of expert systems. Rapid prototyping does not stem from the field of AI. It originates from conventional software engineering as a remedy against the language barrier that frequently shows up in the conversation between end users and system developers. Whatever the value of standardized diagrams and textual specifications for system development and the associated quality assurance, they appeared unable to offer the end user sufficient information (or elicit enough information for that matter) to guarantee the eventual user satisfaction regarding the implementation. This phenomenon can be partly traced back to the end user misunderstanding symbols that are familiar to the software specialist; for another part it can be reduced to the simple fact that it is difficult to estimate just how convenient a certain routine really is without actually trying it. A prototype offers this possibility, as a (simplified) version of the system that is to be developed.

When rapid prototyping appeared successful in finding suitable user interfaces, other applications too, were subjected to it. Hayward[Hayward87] identifies five kinds of prototyping:

- experimental (for finding a suitable user interface or solution strategy);
- exploratory (for requirements analysis);
- performance (for predicting response times, use of memory etc.);
- organisation (for predicting the effects on the organisation);
- evolutionary (for complete system development).

Rapid prototyping is only feasible if the development of the prototype can be accomplished rapidly (inexpensively). This was only possible after fourth-generation languages (application generators) became available. The fact that an expert system shell can in fact be used as an application generator, accounts for the success of these shells as well as for the advance of rapid prototyping as a development method for expert systems. The knowledgebase itself, is then often labelled as a specification of the system. Doyle [Doyle85] remarks that 'with the AI approach (...), the interpreted specification acts as an inexpensive, but full-grown prototype (...) so that this approach easily overshadows others, when formulating the problem becomes complicated - which usually is the case'. Not everyone is (nowadays) biased so positively towards prototyping of expert systems. Breuker and Wielinga [Breuker88a], for instance, referring to [Hayes-Roth83], establish that 'rapid prototyping soon leads to backtracking and discarding systems: in any case, it does not lead to a properly manageable development process'.

The lack of concensus between these experts can partly be explained from the fact that notions get confused: although prototyping is a very valid and reliable aid within a structured development process without endangering the controllability, the way in which (evolutionary) prototyping in expert system development is done, can easily give rise to a chaotic life-cycle and likewise, a chaotic product. This is probably also due to the popularisation of expert systems over the past decade. Every experienced software specialist should know that every program that was subjected to numerous changes should be re-analysed, often be restructured and should sometimes be completely rewritten. Therefore, it is no wonder that an 'incremental evolution' is regarded as a risky working-method [Hayward87].

Doyle too, stresses that the merits of expert system shells should not be looked for in the final product - 'that would probably be more efficient when written in a procedural language, such as C' -, but rather in the fact that a shell enables rapid prototyping and thus substantially reduces costs [Doyle85]. As an example, he gives the PUFF system for diagnosing breathing difficulties. Its development took a lot of testing and reformulating. Yet, less than 50 hours of expert consultation and less than 10 man weeks of programming were needed to obtain a satisfactory system with some fifty production rules.

3.4 Weitzel-Kerschberg life-cycle model

It is expected that a development method for expert systems incorporates a good phasing, leads to clearly defined intermediate and final products, can be applied iteratively, allows for prototyping and provides a well-defined testing and evaluation phase. Though this list is rather ambitious, Weitzel-Kerschberg's method appears a good candidate [Weitzel89]. Important is that this author sees 'systems based on knowledge' as an evolutionary development of conventional systems. It is no wonder therefore, that he is a staunch proponent of so-called 'expert database systems' that are a fusion of database and expert systems.

This development method relies considerably upon techniques such as prototyping, *without* ignoring the development of a conceptual model of the problem domain. In order not to become stuck in sequential development, the individual phases of this method consist of 'processes' that can be *activated*, *deactivated* and *reactivated*. A precondition is that a process can only be activated when the previous process has been activated at least once (with, of course, the exception of the very first process).

A great deal of attention is paid to activities such as refining, redefining and restructuring. The first process, *problem identification*, is concerned with an initial survey of the matter on hand.

Next, a knowledge and information requirements analysis is performed within the *problem definition* process. In close cooperation with an expert, an informal characterization is made of the notions and concepts that play a role in the problem area. The question what purpose the system to be developed should serve is answered here (interpretation of satellite data, displaying data in a head-up display of an aircraft, archiving very scarce knowledge). This phase is also meant to judge the feasibility of the project. This will usually be done by interpreting technical and economical factors. It is expected that this process will stand a good chance of being reactivated later on, especially if additional data are found.

The process of *subproblem identification* subdivides the problem into manageable parts. This leads to a further reduction of the risks involved because now it becomes possible to develop a small prototype in a later stage for every subproblem, that will afterwards be reintegrated.

Concept identification and *definition* record the various concepts and their mutual relationships that are of importance in solving a problem. This process implies amongst others that all objects, attributes, relationships and constraints will be modelled. The foundation is laid for a knowledgebase; what happens here is the establishment of a conceptual model. Knowledge acquisition techniques too, find their place in this process.

In the conceptual design, logical choices are made concerning knowledge representation and databases. The idea that an AI-system adds to the value of a conventional system is true: the knowledgebase is partitioned into data-based and rule-based components.

Detailed design defines the physical system design. Here for instance, the descriptions and pseudo-code of programs are produced. This boils down to mapping the most important concepts, relationships, information flows and solution strategies within a formal representation framework. This formalization-stage eventually results in a prototype that is built with the use of a programming language or an expert system shell.

Actual production of the prototype takes place in the *programming process*. Here, the conceptual design is converted into a runnable equivalent. Regular feedback with the previous process can be expected.

The process of *testing* the *knowledge* and *reasoning capabilities* comes next. Working closely together with the expert, the system is tested. It is attempted to correct evident bugs.

Finally, the *validation* process. The system will now be confronted with a great number of cases concerning the relevant problem area. System performance is then compared to that of the human expert.

The Weitzel-Kerschberg method incorporates the possibility to integrate the development of expert systems into a larger, comprehensive system. The suggested approach can be considered as a separate phase within an SDM-like phasing methodology. At this point conventional and unconventional system development go hand in hand.

4 KNOWLEDGEBASE SPECIFICATION METHODOLOGIES

4.1 Introduction

Purpose of this chapter is the introduction of a method to consistently and completely record the specifications of an expert system. The manner in which a certain knowledge domain of an expert system is specified does not really differ from the way it is done in the development of a database system. This means that the knowledgebase, being the central part of an expert system, is a replication of a knowledge domain in terms of a model. The semantics of the knowledge contained in the knowledgebase must be specified unambiguously. A conceptual scheme of the knowledge is designed, a semantic definition.

Because conventional systems will eventually make use of knowledge modules, it is interesting to take a look first at the possibilities that conventional methods for analysis and design offer to specify these modules. This approach makes integration easier. After that, a description follows of a method specifically developed for the specification of the knowledgebase of an expert system. The chapter will be concluded with the description of an extended conventional method that looks very promising as far as quality assurance of expert systems is concerned.

4.2 Conventional specification methodologies

System development should not be an art. A number of conditions can be attached to the methods and techniques that are used for system analysis and design. They should:

- be independent from the analyst/designer's inventiveness;
- contribute to proper controllability, through division into phases with well-defined products;
- be rational, based on fixed principles so that they contribute to objective decision-making;
- support an iterative development process;
- be transferable, so that design decisions are identifiable and reproducible; there is uniformity in the individual solutions and they are accessible by means of proper documentation;
- be practical (or else remain unused);

- guarantee correctness through design and not through testing;
- be independent from the implementation environment and
- be supported by computer aided software engineering (CASE)-tools.

Most analysis and design methods can be subdivided into a number of categories:
- process-oriented;
- data-oriented;
- object-oriented.

In a process-oriented method, processes are analysed first after which follow the data and data structures, necessary for effectively executing the processes described. The product is a description of the possible processes that alter the states in which a system can find itself.

A data-oriented method gives first priority to analysing the data and data structures and then proceeds with analysing processes. The product is a description of the possible states, state transitions and the description of these transitions.

Object-oriented methods aim at dividing the state description into elementary state descriptions (entities or objects). These entities are obtained by describing the possible sequences of state transitions. Added to that is a description of the dependencies that exist between the state transitions of the various entities.

4.2.1 Structured Analysis

Structured Analysis (SA) is a process-oriented method of analysis [Gane78]. The goal of SA is the development of a logical system model, incorporating graphical techniques that support the communication between users, analysts and designers. First, the problem area is modelled by means of dataflow diagrams. These consist of four elements: information sources/sinks, data-transforming processes, dataflows and data storage. Diagrams are ordered hierarchically, which means that a process itself can in its turn be specified in the form of a dataflow diagram (DFD).

While making DFDs, names for processes, dataflows, data storage and data sources are stored into a so-called data dictionary, which may be seen as a set of definitions at a logical level.

Next, it must be recorded what goes on in the processes. In other words, what functions must be executed by the system. This can be done with the help of decision trees that represent what function must be executed at what moment, or so-called 'structured English', which describes the operation of a function in words.

After that the data storage is analysed. By using techniques like normalization and bubble charting, it is determined how data are stored in a database and what access paths are contained in functions.

An iterative process records the logical DFDs, the data dictionary, the process description and the definition of the database, that all together form the logical specification of the system.

4.2.2 Nijssens Information Analysis Method

Nijssens Information Analysis Methodology (NIAM) is an example of a data-oriented method [Wintraecken85]. NIAM uses three notions:

- knowledge is everything someone knows about a certain subject or a certain area;
- information is knowledge that can be transfered;
- communication is the exchange or transfer of information.

That part of the real or abstract world to which the information exchanged through communication relates is called reality or Universe of Discourse. When we limit ourselves to communicative processes that use an information processing system, then the information has to be represented in the communication medium: these are the data. Communication between humans using an information processing system is only practical if it is defined which data may be used for the communication, and what the semantics of this data is. These matters are described in a so-called conceptual grammar. The information analysis process may now be seen as an activity 'where information, exchanged during communication processes, is analysed and a (conceptual) grammar for these communication processes is formulated, based on this analysis' [Wintraecken85].

In this way, NIAM offers the possibilities to define the specifications in a formal manner. The analysis phase yields a conceptual grammar that is transformed during the design and implementation phase into a computer based information system. The conceptual grammar describes three aspects:

- communication, or information flows;
- transformation, or functions that influence data elements;
- logical data storage method.

Application principles of NIAM are:

- communication takes place by means of natural language sentences;
- communication, transformation as well as the data storage method can be described by means of a conceptual grammar.

NIAM is based upon a number of clear concepts with which specifications can be described. System development along the lines of NIAM consists of three phases. Firstly, the analysis of the activities. This aims to form an activity model and to determine the user needs. With the help of methods and techniques gleaned from the discipline of business administration, the activities can be traced. The results of this phase are usually considered as data during the initial stages of the NIAM analysis. Secondly, the analysis of information. The conceptual grammar must be defined during this phase. Information flow diagrams must be made, representing the relationships between functions, information flows, data storage and information sources/sinks. Top-down decomposition leads to a hierarchical ordering of diagrams. Every flow or transformation is described by means of natural language. After that, the information structure of the flows and data storage must be established. The result is an information structure diagram. The consistency of input and output data in transformation must be guaranteed. Finally it is established what functions must be automated and a definition of the user interfaces after which the construction of the system may begin. Lastly follows the actual construction of the system. Application programs and databases will now be implemented.

4.2.3 Jackson System Development

Jackson System Development (JSD) is an object-oriented specification method that spans the whole life-cycle of a system from analysis through to implementation [Sutcliffe88]. Aim is the development of reliable systems that can be properly maintained by means of concise, unambiguous and (yet) readable specifications. This method differs from other structured methods such as Structured Analysis because it emphasizes modelling and takes the factor time into account.

JSD consists of three phases, with a distinct beginning and end. Project control is simplified by the definition of intermediate products. First, there is a modelling phase. A description of reality is made by recording the actions that are performed and analysing the objects subjected to these actions. By making a list of objects and actions, a definition is obtained of that part of reality that falls within the limits of the system. Actions performed or experienced by an object as well as the order in which this must happen, are structured. In this way, reality is described in terms of actions and objects within a process model.

The second phase is the networking phase. The previous phase distinguishes one or more subsystems, to which, in the networking phase, inputs and outputs are assigned. The output system that is the user interface, will now be analysed as is the input subsystem that takes care of validation. Furthermore, the preconditions are specified as far as time aspects and system processes are concerned.

Finally, the implementation phase. The logical system specification established so far can be seen as a network of parallel, communicating processes, which is transformed into a sequential design using a 'scheduling' technique. This activity is followed by detailed technical design and programming.

JSD starts with the analysis of the most important system structures to form a model of the reality on the basis of this. Only then the functionality of the system will be added. Design decisions of a technical nature are postponed until the final phases of system development.

The principles on which JSD is based are the following:

- before a system can be built it must be understood;
- to come to an understanding, the problem must be modelled;
- systems are always concerned with the transformation of things, therefore, the transformation process needs to be modelled.

The emphasis lies on the deeper structure of the problem rather than on analysis of functions that can be easily observed, as is done in methods such as Structured Analysis. In JSD it is important to know how objects change in time. If the way in which these changes take place is properly modelled, the system should function well. The importance of the factor time is evident. JSD is not directly concerned with modelling the relationships between data, as is NIAM. This must be derived indirectly from the communication process model.

In object-oriented modelling, the system's ability to represent the behaviour of objects in reality is important. A resemblance to JSD is clearly present. Another innovative element of JSD is the view that a system can be considered as a cluster of subsystems that exchange messages. If a separate system process must be created for each subsystem, it is strongly reminiscent of new computer architectures where each process has its own processor available, that, in its own turn, is connected to other processors. The specification can then be represented in hardware.

4.3 AI specification methodology

4.3.1 Knowledge Acquisition, Documentation and Structuring

The adherents of Knowledge Acquisition, Documentation and Structuring (KADS) [Breuker87] oppose the traditional AI approach to knowledge acquisition prototyping. Breuker and Wielinga maintain that prototyping, because it forcibly imposes abstraction, produces an unstructured knowledgebase without the conceptual insight necessary for explanation and maintainability [Breuker88a]. They claim that this holds true especially for knowledge domains of non-trivial complexity or size.

[Hayward87] compares the use of prototyping versus the use of KADS with 'craft versus industry'. He considers the experimental development of a new type of software as a kind of ingenious handicraft. As soon as this technology has matured and commercial production has been started, then the software-manufacturing point of view is more appropriate. The exploratory approach must then be abandoned in favour of a more structured approach that can be controlled better.

The innovative angle of KADS lies in the fact that it promotes choosing an epistemological framework as a first step in knowledge acquisition. Subsequently, one tries to place the accumulated knowledge inside this framework. The framework chosen, interpretation model in

KADS, is a domain-independent abstraction of domain-dependent (but implementation-independent) conceptual models.

Breuker and Wielinga name a number of knowledge acquisition shells as forerunners of KADS because they impose an interpretation model, such as Roget [Bennett85], MOLE [Eshelman87] and SALT [Marcus87] as well as TEIRESIAS [Davis82] and OPAL [Musen87] that are used for MYCIN and ONCOCIN, respectively. But at the point, however, where these knowledge acquisition shells force the knowledge engineer to use a single interpretation model that is part of the shell, KADS offers the possibility to alter the interpretation model completely or in part if it appears that it is not (at all) satisfactory.

For this purpose, a typology of generic tasks has been established. A generic task is a (domain-independent) abstraction of an inference process that may be seen as a conceptual entity. Clancy's heuristic classification is an example of an elementary, generic task [Clancey85]. KADS task typology sees heuristic classification as a form of singular diagnosis, that in its turn is a form of diagnosis etc. An assumption of KADS therefore is that inference processes can be abstracted within a specific domain (i.e. stripped from its domain-specific notions) that can still be described clearly enough to recognize an equivalent as such in another domain. The subjective impressions of those that have used KADS in practice, partially support this but more substantial indications are lacking [Breuker88a].

Ideally, an interpretation model consists of an inference structure and a task structure. The first defines what the primitive inferences are (in terms of name, inputs, outputs and a very broad description). The task structure defines how these basic inferences should be linked together to execute the task in question. Unfortunately, KADS has failed to work out the task structures of interpretation models. The task structures used in actual applications appeared to be too diverse. In point of fact this means that KADS interpretation models purely consist of inference structures like that shown in figure 4.1, with concise textual information added. Primarily, they have a declarative (non-procedural) character. An interpretation model purely indicates what type of objects (meta-classes) function as input or output of the appropriate types of primitive inferences (knowledge sources) and does not say anything about the order in which these knowledge sources must be activated, nor the conditions under which this happens.

The universal structure that KADS imposes on all conceptual models is a four-layer organization. The domain layer contains knowledge about objects (concepts) of the actual domain and their interrelationships. The inference layer specifies what inferences are possible. The task layer specifies in what situation which inferences should be made in what order. The strategy layer, finally, contains the knowledge that is necessary to switch to another solution strategy if the present is unsatisfactory. Breuker and Wielinga remark that this layer does not or barely present itself in reality [Breuker88a].

4.4 A comparative analysis

Until now, no satisfactory specification language has been found for inference structures in expert systems. The diagrams used in KADS as a representation of interpretation models are too ambiguous to act as a specification language and so cannot effectively bridge the gap between the raw output of knowledge elicitation and implementation. However, the selection of a suitable interpretation model as well as the transition of a conceptual model to an implementation, must go smoothly if KADS is ever to be proposed as a commercially applicable development method.

The most important component of the NIAM methodology is the transparent representation formalism which makes it possible to generate procedures and files from the specifications by means of an application generator [Nijssen89]. This is, as indicated above, not yet possible using KADS. Especially database experts suggest regularly to use the relational database paradigm for expert systems. The NIAM data models can be useful for domain structuring, something that has not been worked out sufficiently yet in KADS [Breuker88a]. NIAM, as described in [Wintraecken85], concentrates on data structures and their integrity. Less attention goes to defining procedural knowledge. The potential of Structured Analysis [Gane79] as well as that of single-model knowledge acquisition tools like MOLE [Eshelman87] and CSRL [Bylander86] merit, beside NIAM, further investigation as far as this point is concerned.

The question in how far methods such as SA, JSD and NIAM are suitable for specifying a knowledgebase cannot be answered completely. Until now, not many cases are known where one or more of these methods have been applied building intelligent systems. However, in the research literature there is much speculation about this subject.

[Vogler88] states that the knowledge levels of KADS are relatively easy to transpose to SA. The domain level is similar to the data-dictionary that is compiled, the inference level can be compared to a data flow diagram and the task level resembles the definitions of an SA process structure. An advantage of this approach is that the integration is far easier in conventional system development. A disadvantage is that the SA method remains very process-oriented. This houses the danger that a specification of the domain level is far more difficult to realize. And it is especially this level that is essential to an expert system and has been ignored in the past for too long.

There are no known examples (yet) of an expert system developed with the use of JSD. JSD strongly emphasizes the modelling of objects in reality (as far as they lie within the limits of the system that has to be developed) and the actions to which they are subjected.

NIAM offers excellent possibilities to specify the domain level of an expert system consistently and completely [Bruin88]. This method that is data-oriented, contrary to SA and JSD, is used among others for the development of a conceptual model for database systems. As said earlier, NIAM allows for the definition of so-called constraints. These rules restrict the occurrence of certain facts in an automated information system. Especially [Nijssen86] explores the hypothesis that an expert system is nothing but an automated information system containing certain human expertise. This expertise (or knowledge) is only a set of interrelated facts. Facts may be entered by the user (as the result of a query or during the creation of the database). It is also possible that they are derived from other facts that are already present in the database, with the help of derivation rules. Finally, they can be the result of a reasoning process: by applying inference rules on facts already present, new facts can be inferred.

[Breuker87] makes a subdivision into domain, inference, task and strategic levels. Each of these levels takes a particular look at knowledge: the domain level may be described as the most concrete because here objects in the problem domain and their mutual relationships are defined. The strategic level approaches the problem domain far more abstractly. Indeed, the KADS theory at this point is so abstract that this level is never used in practice. The intermediate levels (inference and task) contain the dynamism of the expert system as it were. The inference level indicates what elementary inferences (specification, abstraction etc.) can be performed with the objects and relationships at the domain level. The task level groups (and activates) inferences in order to define a certain task (classification, diagnosis etc.).

Undoubtedly, the specification of the domain level has a strong resemblance with the specifications that are defined during the development of a database system. Here too, a data dictionary must be built: a file that records character and nature of each entry that is used (definition, significance, physical implementation and relationships with other entries). The inference level can be seen as more process-oriented. By indicating what exactly is the input and output of elementary inference steps (processes), a functional model of the problem domain can be made. The task level makes the model dynamic, because with the help of process specifications it describes what action(s) must be executed. Because of the minor practical relevance, the strategic level will not be dealt with any further.

The constraints, that can exist in the form of derivation rules or inference rules, are of great importance. By choosing a method that offers a good support, not only at the domain level but also at the more abstract levels of knowledge, the experience that was accumulated in the database community with regard to the integrity problem can be transposed to the knowledgebase community.

The next paragraph presents an extended version of NIAM (ExtendedNIAM) that offers better features. ENIAM appears to be a method worthy of ample consideration in vast C^3I systems, equipped with intelligent modules.

4.5 Knowledgebase specification

4.5.1 Specification requirements

An important activity during expert system development is the definition of a conceptual model. The conceptual model must be a consistent representation of the knowledge domain in which, just like in database applications, a distinction is made between a knowledge scheme (the definitions of all existing facts and relationships) and the actual knowledgebase. The advantage is that now the knowledgebase can be discussed in abstract terms. Consistency and completeness can be controlled by means of constraints that are imposed upon the actual knowledgebase. A conceptual model must be made using a method that explicitly defines the distinction between facts and their definition. The previous paragraph labels NIAM as a prominent candidate in this respect. Meanwhile, colleagues of Nijssen, the developer of NIAM, have added extensions to this method

[Creasy89]. The core of these is formed by the possibility to formally represent constraints (e.g. derivation and inference rules) in what is called E(xtended)NIAM. This is the first step in the direction of a more extensive integrity control within this method because the representation of constraints makes it 'simple' to generate executable Prolog programs. These programs (being the exemplification of the constraints) can be, with the help of algorithms that will be dealt with later in this study, tested for consistency and completeness. This aspect has already been the subject of investigation by [Creasy88].

Based on [Wintraecken85] the specification must satisfy certain requirements. It must be in accordance with the:

- conceptual principle;
- 100% principle;
- natural language principle.

Conceptual principle

A conceptual grammar is a description of all rules that prescribe what situations and transitions between situations may occur and what the meaning is of the data that are stored in the knowledgebase. The grammar consists of a specification of:

- elementary sentence types to which elementary deep structure sentences must belong, that describe the contents of the knowledgebase; this implies the use of the notions object, relationship and constraint;
- constraints.

The grammar describes only the conceptual aspects of the information exchange with the expert system, i.e.:

- describe only the meaning of the data;
- no realization aspects, i.e. implementation-independent;
- no description of the internal / external form of representation;
- indicates what exclusively must be described by the grammar.

100% principle

The grammar describes all conceptual aspects of the knowledge exchange with the expert system:

- the meaning of all data is described in the grammar;
- no other data are exchanged with the expert system other than those described;

- when interpreting data, users may only use rules from the grammar;
- never use the knowledge about reality, implicitly present in the users; describe everything explicitly.

Natural Language Principle
Knowledge can always be described by means of elementary deep structure sentences of a natural language:
- the grammar is described by means of a natural language;
- all implicit knowledge is made explicit.

The conceptual principle takes care of the consistency and completeness of specifications. The completeness is guaranteed by the 100% principle. The use of a formal language for defining the specifications leads to conciseness. Nevertheless, the readability is 'good' because the principle of natural language is used. Portability implies independence from implementation-dependent matters. Here too, the conceptual principle is important. The adaptability remains to be discussed. The conceptual principle and the 100% principle are conditional for this.

4.5.2 Constraints

Vital to NIAM (and ENIAM) is the principle of describing reality with the help of a natural language. In the development of expert systems, use is made of knowledge/information that has been recorded in writing (manuals etc.) on the one hand, and on the other hand, knowledge that is based more on experience that has been derived from an expert through interviews. The intention of knowledge acquisition is to add a structure to this jumble; a conceptual model of the knowledge domain must be designed.

Reality consists of a set of objects that are relevant within a certain knowledge domain. This thesis, based on [Wintraecken85], implies that information or communicable knowledge is nothing but making a statement or an assertion about objects within this reality. A fact is the elementary assertion that establishes a relationship between objects in this reality. A feature of an elementary assertion is that it only contains a single fact. Every non-elementary assertion can be broken down into elementary assertions.

The objects in an elementary assertion can be divided into lexical and non-lexical ones. Lexical objects can somehow be represented in the form of letters, numbers and/or other symbols and may be seen as names or references to other objects. Non-lexical objects cannot be represented. The elementary assertion: 'Jones is the family name of an employee' contains the name 'Jones' as a lexical object and 'employee' as a non-lexical object which is referenced by means of the name.

NIAM substantiates reality with the help of binary facts, i.e. facts in which two objects from reality always play a role. A distinction is made between two types of binary facts: bridges and ideas. A bridge is a fact that relates to a lexical and a non-lexical object; it forms, as it were, a bridge between two worlds. An idea is a fact that relates to two non-lexical objects; the name idea was chosen because non-lexical objects are discussed without mentioning the lexical objects that reference them. The example in the last paragraph is a bridge.

One can also speak of types: object types (lexical and non-lexical) and fact types (idea and bridge types). Introducing this abstraction, it becomes possible to refer to a set or class of equivalent elements. The assertion: 'There are family names' is an example of the specification of a lexical object type. The assertion 'There are employees' specifies a non-lexical object type.

NIAM is based on the assumption that knowledge can be described with facts and rules. Facts can be seen as knowledge concerning a certain reality that may change in time. Rules are elementary assertions that are no facts; they define which facts can be part of communicable knowledge and which cannot. All facts are incorporated in the database, whereas the grammar contains a description of all the rules.

An important type of rules are the constraints. These are rules that are no specification of object types or fact types; they restrict the facts that are allowed on the basis of the definition of object types and fact types. Constraints, in their turn, can be subdivided in static rules and transition rules. Static rules define which facts may be present in the database at any given time; they describe states. Transition rules define which changes are allowed in the database; they describe transitions from one state to another.

In general, the information/knowledge analysis with the help of NIAM is conducted along the following lines. Reality must first be described in words. Then, this description must be unwound into elementary assertions. Subsequently, these elementary assertions are divided in facts and rules, and are described formally.

NIAM does not make a distinction between graphic and non-graphic constraints. The first can be represented in information structure diagrams (ISD) using certain symbols. The latter must be added to an ISD by means of text.

A graphic constraint, for instance, is the unicity rule that indicates whether there is a 1:1, 1:M or N:M relationship between objects. Non-graphic constraints can be divided into value rules, cardinality rules, rules determining subtypes, derivation rules, transition rules and inference rules.

4.5.3 An extension of NIAM: E(xtended)NIAM

An important category of constraints is the non-graphic category. They share the property that they cannot be graphically defined by the information structure diagrams (ISD) of conventional NIAM. Normally, they are added to the graphic representation in natural language. This is in some respects, a disadvantage since an ISD is more inviting to the user (domain expert) during validation of the conceptual model than is a piece of text. In the application of NIAM during the development of a knowledgebase, it is necessary to use non-graphic constraints. Especially inference rules are quite similar to what is called production rules in other methods. Referring to the KADS method [Breuker87], conventional NIAM can offer a better support at domain level than the usual AI development methods. Inference and domain level can only be partly covered by non-graphic constraints.

The advantages of the NIAM method are the clear distinction made between type and instances, and the fact that (a number of) constraints can be defined graphically. Without compromising these points, extended NIAM must be able to add more semantics to the information structure diagrams. [Creasy89] has invented an adapted method called E(xtended) NIAM. This method not only offers the possibility (as does NIAM) to define 'set-oriented' constraints graphically, but also 'member-oriented' ones. Examples of constraints based on sets are unicity, exception, and totality rules. A member-oriented constraint makes a statement about the occurrence or nonoccurrence of a member within a certain set.

The extension is based upon existential graphs [Sowa84], a graphic representation of logical statements. A small number of symbols is used that has the same expressive power as first order logic. The existential graphs consist of two parts, alpha and beta. The alpha part contains three basic elements, a sheet of assertion, a graph and a cut. The sheet of assertion may be seen as a model of reality. An instance of a graph is equivalent to a proposition (a statement about reality). Graph instances are shown on the sheet of assertion and are considered to be true. The cut is a negation of a graph and is represented as a line drawn around the graph. Examples of the alpha part of existential graphs are shown in the next figure.

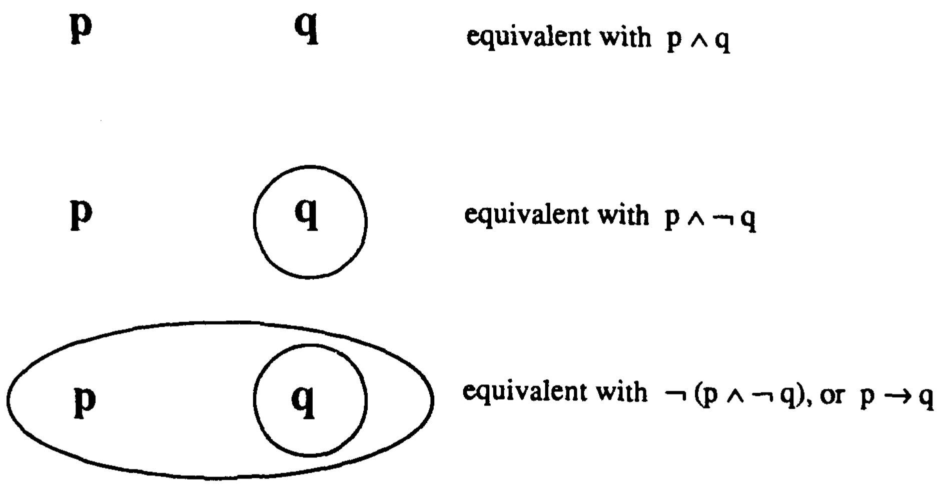
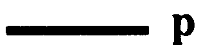
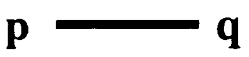

p q
equivalent with $p \wedge q$
p q
equivalent with $p \wedge \neg q$
p q
equivalent with $\neg (p \wedge \neg q)$, or $p \rightarrow q$

Figure 4.1: Examples of existential graphs (alpha part).

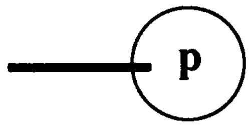
It appears that existential graphs implicitly contain only a few logical operators, i.e. conjunction and negation. There is, however, another element that stems from first order logic, namely the existential quantification $\exists$. By means of a line of identity, individual objects in reality are depicted. The beta part is based on predicate logic approach, whereas the alpha part is based on proposition logic. Examples can be found in Figure 4.2.



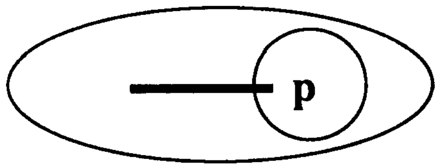
equivalent with $\exists X\ p(X)$



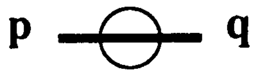
equivalent with $\exists X\ (p(X) \wedge q(X))$



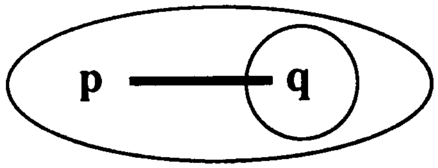
equivalent with $\exists X \neg\, p(X)$



equivalent with $\neg\, (\exists X \neg\, p(X))$, or $\forall X\ p(X)$



equivalent with $\exists X\ \exists Y\ (p(X) \wedge q(Y) \wedge X \neq Y)$



equivalent with $\forall X\ (p(X) \rightarrow q(X))$

Figure 4.2: Examples of existential graphs (beta part).

4.5.4 Specification in E(xtended)NIAM

From the previous paragraph it appears that it is relatively easy to define an inference rule (IF condition THEN action) with the help of ENIAM. In this paragraph it will be demonstrated, using a small example, that (recursive) constraints can be represented with ENIAM ISDs.

This may be illustrated with the help of the ancestor problem. Solving this involves as a first step, determining when person A qualifies as an ancestor of person B. There are two distinct cases:

1. A is an ancestor of B if A is a parent of B;
2. A is an ancestor of B if A is a parent of X and X is a parent of B.

If one has at one's disposal a set of facts defining who is the parent of which person, then one can determine the set of ancestors of any person by applying a recursive rule. Defining a derivation rule as is the case with case 2, will not be a problem with ENIAM because it is possible to discuss the individual elements of a set. The ISD is shown in figure 4.3.

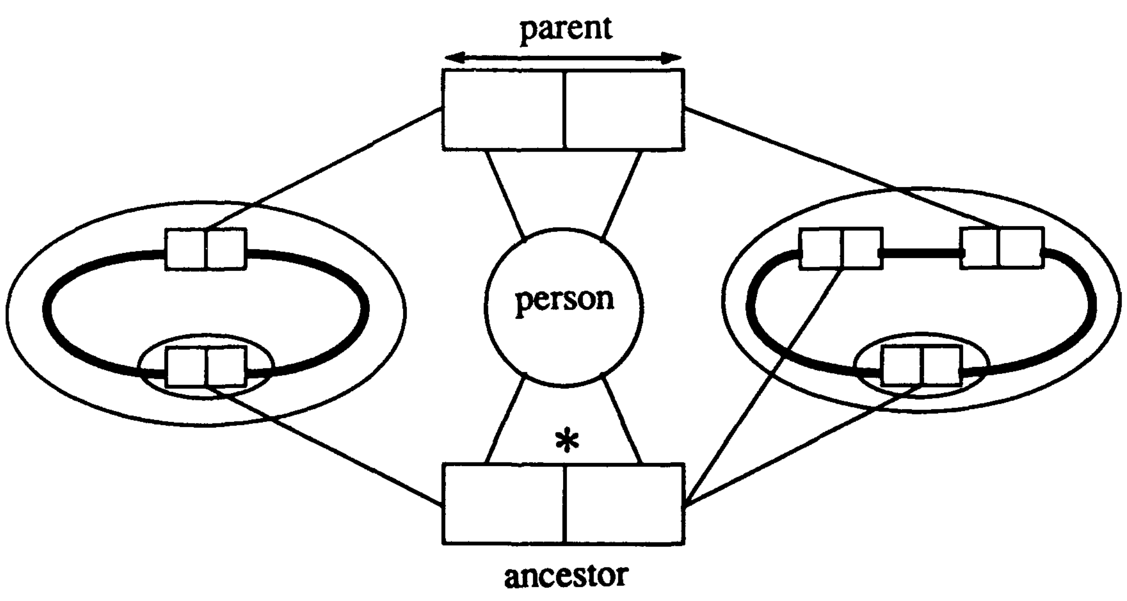


Figure 4.3: **ENIAM information structure diagram for the ancestor problem.**

The arrow over fact type Parent and the asterisk over Ancestor have the same meaning as in NIAM, a M:N relationship and a derivable fact type, respectively. According to NIAM conventions, the ISD should exclusively consist of the object type Person and the fact types Parent and Ancestor. The two rules mentioned before that give a definition of the notion ancestor, could, in NIAM, only have been added to the ISD as a small piece of text. ENIAM allows graphic representation with the extra advantage of each and every constraint being defined within one and the same formalism. The general structure of an if-then rule is depicted in the enlargements on the left and right side of the original ISD. The existential graph on the left contains both fact types Parent and Ancestor as in case 1. Referring to the last example of figure 4.2, p stands for the fact type Parent and q for the fact type Ancestor. There are two lines of identity because binary fact types are involved here. Each of these denote a single member of a certain set. The left part of the ISD in figure 4.3 is equivalent to:

$$\begin{aligned}&\forall \text{PersonA, PersonB:}\\&\qquad (\ \text{parent(PersonA,PersonB)} \rightarrow \text{ancestor(PersonA,PersonB)}\ )\end{aligned}$$

The fact type Parent occurs once on the right side whereas Ancestor occurs twice, according to the second (indirect) case. This is equivalent to:

$$\begin{aligned}&\forall \text{PersonA, Intermediary, PersonB:}\\&\qquad (\ \text{parent(PersonA,Intermediary)} \wedge \text{ancestor(Intermediary,PersonB)}\\&\qquad\qquad \rightarrow \text{ancestor(PersonA,PersonB)}\ )\end{aligned}$$

When these two, logical expressions are joined together with the help of a logical or-operator, the following expression results, completely defining the ISD in figure 4.3:

$$\begin{aligned}&\forall \text{PersonA,Intermediary,PersonB:}\\&\qquad (\ \text{parent(PersonA,PersonB)} \vee\\&\qquad\ \ \text{parent(PersonA,Intermediary)} \wedge \text{ancestor (Intermediary,PersonB)}\\&\qquad\qquad \rightarrow \text{ancestor(PersonA,PersonB)}).\end{aligned}$$

The logical expressions given in this paragraph can be simply transformed into an executable formalism, i.e. the Prolog programming language. Each of the two cases in which the ancestor problem can be broken down, lead to a line of Prolog. The or-relationship that exists between these lines is taken care of by the interpretation mechanism of Prolog; if the first line, after matching the arguments, does not lead to success, the second will be tried. During the direct transformation of an ENIAM specification into Prolog, fact types are expressed by binary predicates. The logical expressions given before only have to be duplicated in reverse. This is shown in figure 4.4.

```
ancestor(PersonA,PersonB):-
        parent(PersonA,PersonB).

ancestor(PersonA,PersonB):-
        parent(PersonA,Intermediary),
        ancestor(Intermediary,PersonB).
```

Figure 4.4: Ancestor problem in Prolog.

First, an executable version of Prolog is made, to be able to check the integrity of the conceptual model of the ancestor problem. The consistency and completeness of this program, and indirectly, the conceptual model, can now be investigated using the algorithms of chapter 6.

5 THE EXPERT SYSTEM AS A PRODUCT

5.1 Introduction

Because the subject of product quality comprises too many aspects to fully investigate within the scope of this project, a selection has been made from various quality criteria. The selection consists of integrity, testability and maintainability, given concrete form in constraints for consistency, completeness and other forms of integrity as well as modularity, respectively. Integrity constraints may be seen as a counterbalance for the poor testability and can contribute to proper maintainability. Modularity plays a role in testability as well as maintainability.

The selection of the issues mentioned above does not mean of course that the others are not important. They are, however, somewhat more difficult to handle without coupling them to specific domains (e.g. computational efficiency) or performance specifications (e.g. fail-safe for real-time systems), or they are less important to the quality problem in expert systems. Portability, for instance, is an aspect that is not or just barely concretized in expert systems but in software. User friendliness (quality of the user interface) is an aspect that by many is considered to be most important. Matters like hypertext and user models can greatly enhance the quality of these [Conklin87, Schneider84]. It can, however, hardly be seen as expert system specific, apart from the intrinsic quality of the explanation that must be generated by the system. As far as domain-independent guidelines can be given for this, these result in an explicit representation of meta-knowledge, which can be seen as a form of structuring or modularity [Wognum89]. Modularity is a a characteristic of software that is very important while studying expert systems. This study looks at it from the angle of testability, because integrity results from it.

5.2 Testability of expert systems

5.2.1 Conventional testing methodologies

Methods to guarantee the reliability of software can be subdivided into methods that 'embed' (enhanced) reliability (e.g. structured analysis, design and implementation methods) and methods that test reliability afterwards. The overlapping notions testing, evaluating, validating and verifying belong to this last group. Over the years, an extensive range of methods, paradigms and hints has been developed for testing software. Starting principle of this paragraph is a testing method as described in [Myers79]. Whereas this method is very useful for testing conventional programs like interpreters and operating systems, it seems to be inadequate for expert systems and (to a somewhat lesser extent) other data-dominated systems such as databases [Deutsch82]. It remains a question by what it is caused. To get a better insight into this issue, we will start to review some techniques of Myers' testing method. Testing, the Myers-way, must partly be done by executing the software and partly by using one's brains.

To the mental techniques belong, among others, inspection and walkthrough. Walkthrough comes down to investigate how the interpreted program will react to certain test cases using the source code. As in every form of testing, the expected or desired output must be specified before actual testing. Inspection assumes two different forms: a verbal paraphrase of the programmer of what the program does (using the source code) and running down a check-list with likely errors. The check-list given in [Myers79] contains, for instance, items such as 'index within bounds?'. This is of little use when one programs in a language like PASCAL: if a reasonably well-designed compiler is used, a possible 'index out of bounds' error will be detected automatically during compilation.

Testing through execution also consists of two types, i.e. black box and white box testing. The first type considers the system or module to be a black box: only the result (the output) is important when feeding certain input. How this software establishes its output is of no importance to black box testing. Because it is virtually impossible for most software to try all possible input values, it is attempted to compile a set of test cases that is as efficient as possible. Efficient here means: a small number with a large probability of errors manifesting themselves. Experience teaches us that the value range of input can be divided into so-called equivalence classes, characterized by the fact that two test cases from the same equivalence class will give rise to the same errors(if resulting in an error at all). Of course, the determination of equivalence classes is a gamble in case of a black box: only the specification gives a lead. Yet, we can make an educated guess on the basis of that. To select , for example, equivalence classes for a module that, according to a (concise) specification, 'extracts the root of a number' one could reason as follows. One can distinguish the subsets $x | \sqrt{x}$ not defined as $x | \sqrt{x} >= x$ en $x | \sqrt{x} < x$ as far as the function that x adds to the root of x within the real numbers. Besides that, one can make a distinction between integers and fractions (the distinction between rational and real is not important of course: normally, only rational numbers can be represented). Besides the notion of equivalence class, the notion of limit plays a role: it is good, heuristic practice to use test cases with values on the very limit of their equivalence class. In the case on hand, this could lead, for example, to the set of test cases (-1, ?), (0, 0), (0.01, 0.1), (1, 1), (2.25, 1.5), (1.0E30, 1.0E15). Because of the heuristic character of the use of equivalence classes and limits it cannot be determined objectively, what exactly is the 'best' set of test cases. Furthermore, an optimum state is always a balance between enhancing the error-finding probability and limiting the time involved in testing.

Contrary to black box testing, white box testing uses the structure of the source code. The thought behind the white box techniques presented by Myers, is that the ideal set of test cases will literally cover every nook and cranny of a program. This ideal can best be approached using path coverage. In path coverage, every possible path of execution in the program is covered by (at least) one test case. In general, the demand for a path coverage leads to unacceptable costs (it must be kept in mind that a path that runs through one or other repetitive loop differs from the path that runs through one and the same loop for three times). Therefore, one has established a set of coverage criteria, that will be discussed in order of increasing importance.

a. *Statement coverage:* Every statement must be executed during testing at least once.

Example: To test 'IF c1 OR c2 THEN a1 ELSE a2' with a statement coverage, you need at least two test cases: one executing a1 and a second executing a2.

b. *Decision coverage*: Every branch (the result of a decision) must be run through during testing at least once.

Example: To test 'IF c1 OR c2 THEN a1' with a decision coverage, you need at least two test cases: one resulting a 'TRUE' for 'c1 OR c2' (and thus executing a1) and one resulting in FALSE. Note that one test case would suffice as a statement coverage. Decision coverage implies statement coverage.

c. *Condition coverage*: Every condition (term in a logical expression) must at least produce one instance of TRUE and one of FALSE during testing.

Example: To test 'IF c1 OR c2 THEN a1 ELSE a2' with condition coverage you need at least two test cases; one for instance, with c1 resulting in TRUE and c2 in FALSE and one producing the reverse. Note that in this case the requirements for decision coverage or even statement coverage are not satisfied.

d. *Decision / Condition coverage*: Both decision coverage as well as condition coverage must be satisfied.

Example: To satisfy the aforementioned example two test cases will satisfy, i.e. one with c1 and c2 both qualify as TRUE and one where both yield FALSE. Decision / condition coverage is a far better criterion than condition coverage: the test set covers more of the program, but need not be larger!

e. *Multiple condition coverage*: During testing, the tuple of conditions in a decision must be run through every possible value at least once.

Example: To test 'IF c1 OR c2 THEN a1 ELSE a2' with multiple coverage you need at least 2 to power of 2 = 4 test cases, one with both conditions resulting in TRUE, one making both FALSE and two with one resulting in FALSE and the other TRUE. Multiple condition coverage implies decision / condition coverage but, in general, requires more (and sometimes considerably more) effort in testing. Multiple condition coverage does not imply path coverage in any respect, not even in a program without repetitive loops (a decision tree). All possible combinations of condition results would be tested for each, individual decision. Path coverage would imply that all possible combinations of decision results (per program) would be tested.

It is reasonable to assume that non-nested decisions can be examined independently from each other. This is not true, however, in the case of nested decisions, especially when multiple condition coverage is required. [Myers79] does not give a clear statement on this. Nevertheless, it is repeatedly indicated that multiple condition coverage is a necessary prerequisite for proper testing. When you consider that nested decisions may be seen as a single decision, a maximum number of conditions of about 20 would not seem impossible for extensive applications. This would come down to a magnitude of about a million test cases. It is therefore safe to assume that conditions in nested decisions do not influence each other or at least not always.

5.2.2 Applicability of conventional testing methodologies on expert systems

Considering the statements in various publications about the intricate flow of control, impairing proper testing of expert systems, and the 'complications during the testing of data-dominated systems', it is concluded that the specific problems inherent to testing an expert system belong to the *declarative* part (the knowledgebase(s)) and not to the inference engine (shell) [Llinas87 Myers79, Deutsch82]. It is held, in general, that the advantage of explicit (declarative) knowledge representation over implicit (procedural) lies in the fact that shell and knowledgebase may be tested separately. During the discussion of the applicability of the aforementioned white and black box testing methods, it is assumed that the shell will contain no bugs (anymore).

Equivalence class, the leading concept in black box testing can only be useful in a specification that is either very detailed or will be approached identically by different persons. In black box testing it is also desirable that the number of test cases necessary is not too large. [Doyle85] remarks that 'in the AI approach (...) the interpreted specification functions as an inexpensive, but complete prototype'. Here, the knowledgebase is considered to be a (sub)specification that together with the (conventional) specification of the shell, forms a specification of the expert system. This may be true when the knowledgebase is relatively small and orderly, but should this not be the case then it seems inappropriate to designate the knowledgebase as a specification. A requirement that must part of every specification is that its validity can be checked without problems. Should the knowledgebase be poorly organized, however, then this will not be possible. The main question now is whether in such a case a specification of the knowledgebase can be formulated that can alleviate the testing and maintainability problems observed, or eliminate them altogether. After having read the previous chapter, the reader will not be amazed that our answer to this question ranges from a cautious (as far as elimination is concerned) to an unconditional (as far as alleviation is concerned) yes. Doyle's statement cited previously seems to hold true for the *rule* part of an expert system. In such a situation black and white box methods coincide. The only way then, to enhance the orderliness of product *alias* specification is to reformulate the rules, using, for instance, other domain concepts or a totally different approach producing more modularity. It goes without mention that generally speaking, this will not be a simple task.

The use of a limit in test cases will in general be less fruitful than in expert system, simply because the variables are symbolic more often and thus discrete or even binary. Illegal values, however, can play a very useful role, especially there where making test cases gets complicated because the maker is not able supply the desired (correct) output to the input. Take, for instance, an influence graph of physiological variables with observable phenomena at the extremes (symptoms or syndromes). Usually, one has to be an expert (a physician) to come up with a (minimal) common cause of the symptoms of hypertension (high blood pressure) and haematuria (blood in urine). A grain of common sense suffices, however, to infer that a combination of coma and hyperventilation is illegal. The test case (coma, hyperventilation,??) can thus be made without the help of an expert.

(This test case is derived from daily practice; it immediately demonstrated an essential design error in the respective algorithm). When an expert system is subjected to multiple condition coverage (about nested conditions), then this will soon lead to an unacceptably high number of test cases. The fact that an expert will usually have to assist in connecting the desired output to the input of the test cases makes it even worse. After all, the availability of experts is a familiar bottle-neck in the development of expert systems.

Holding on to limited multiple condition coverage (i.e. within a decision) is, however, dubious in many cases, especially in the case of a decision tree. A subdivision (preferably hierarchically) of the knowledgebase in small modules is therefore desired. Multiple condition coverage can then be demanded in every individual module. If these modules are not too large and legitimately claim the title of module, in other words, form conceptual units without too much interaction, then such an approach is undoubtedly optimal. Which demonstrates the vital importance of structuring and modularity to testability. Needless to say perhaps, is the remark that the aforementioned list of requirements has been written down without taking into account its feasibility. It is merely a sketch of the structure that would be *ideal* for testing and maintenance. In practice things would be too rigid (in hierarchical structuring, for instance) to serve as an effective guideline.

5.3 Structure and modularity

5.3.1 Modularity in conventional software

In a traditional sense a software design is structured properly, when the modules possess a certain independence from each other and every module is internally coherent from a functional point of view, i.e. actions within the module are related to each other. The first aspect is called coupling, the second is called cohesion. The concepts are not mutually independent. As modules obtain a stronger cohesion, generally a weaker coupling will suffice to let the whole system function properly.

Coupling is the degree of dependence between two modules. This rather abstract definition can be operationalized as the probability that during encoding, debugging or modification of a module, the contents of another module must be taken into consideration. The coupling between modules comprises the following four aspects:

a. *Nature of the coupling between modules*
A coupling through sending messages and their reactions as is used in object-oriented programming, is of an entirely different nature than the mutual use of a global data area (common environment coupling). The coupling through import and export of variables and subroutines lies more or less in between.

b. *Complexity of information exchange*
This must be measured in terms of the *number* of items that is exchanged between modules; not the *amount* of data exchanged expressed in bytes.

c. *Time of instantiation of a connection between modules*
Connections instantiated at compile time are preferred to those instantiated at run-time.

d. *Nature of the information exchanged*
On the basis of this, [Gane79] distinguishes three types of coupling, i.e. data-coupling, control-coupling and content-coupling. The latter is also called external or pathological coupling.

The significance that [Gane79] gives to these three terms, essentially means that data-coupling must be preferred to control-coupling because the first does not influence the flow of control of the receiving module and the second one does. The transfer of flags comes to mind in the latter case. Content-coupling is inadmissible altogether. This means that a module covertly reads or changes the value of a variable in another module. One can think of a module hierarchy, for instance, that is being skipped surreptitiously.

The distinction between data and control-coupling is formulated somewhat extremely. It is unavoidable that the flow of control of the receiving module will be influenced. After all, in every subroutine that tests the value of an argument (e.g. in a subroutine that extracts roots, a test that determines whether or not the argument is negative) it can be maintained that the value of the argument influences the flow of control. This form of control-coupling is not all that inferior to pure data-coupling as long as two conditions are met. Firstly, it must be possible to specify the influence of information coming from the calling module on the flow of control of the called module, without referring to the identity of the caller. Secondly, the influence may not have any effect on the flow of control in the module after the result has been given back to the calling module.

As has been remarked before, a surplus of coupling between modules can generally be taken as the result of a lack of cohesion within modules.

Usually, in software one can discern six to seven levels of cohesion, i.e. (in decreasing order of preference):

a. *Functional cohesion*
   Every element (subroutine) in a functionally coherent module forms an integral part of a single function. In other words, the elements are a unit, procedurally as well as *conceptually*. As a test for functional cohesion, one can use the rule of thumb that it must be possible to express the function of a module in the imperative plus a direct object (e.g. 'calculate the optimum solution' or 'format the answer').

b. *Sequential cohesion*
   A sequentially coherent module is a cluster of a number of functionally coherent parts where the output of one part serves as input for the following part (e.g. 'read line of input and delete spaces' or 'calculate solution and print it').

c. *Communicative cohesion*
   A communicatively coherent module consists of functionally coherent components that have a direct object in common (e.g. 'print the solution and store it in the database').

d. *Procedural cohesion*

Procedurally coherent modules can emerge when the symbols of a flowchart are placed into a module (e.g. process symbols in data flow diagrams). When this results into a conceptual unit this is called procedural cohesion.

e. *Temporal cohesion*

The elements (subroutines) in a temporally coherent module have in common that they occur in the same stage of execution (e.g. initialization).

f. *Logical cohesion*

Similar looking pieces of code yet being different in detail that are merged with the exclusion of identical elements (except for one item, of course) form a logically coherent module (e.g. a module containing all input procedures).

g. *Coincidal cohesion*

Coincidally coherent modules consist of elements that have no common denominator other than contained within one and the same module.

5.3.2 Modular structure of knowledgebases

The traditional knowledgebase consists of a tangle of production rules without any form of modular structuring. The emergence of second generation expert systems has started a first incentive to structuring: meta-knowledge is separated from domain-knowledge. Gradually, the knowledgebase programmer looses the freedom that was a former pitfall. The need of prestructuring the knowledge model as a specification of the knowledgebase is generally recognized nowadays. There is, however, no concensus yet about the form such a prestructuring should have. The translation too, of a structured specification into a knowledgebase structure usually lacks a clear description.

Within the framework of the KADS project, a three to four-layer knowledge model is used, i.e. a domain-layer, a (canonical) inference-layer, a task-layer and sometimes (seldom in concrete applications) a strategy-layer [FEL1989-148]. The central concept of KADS is that of a generic task. As a good example we mention the heuristic classification task [Clancey85]. KADS, however, aims primarily at knowledge acquisition. It pays too little attention to other relevant matters to qualify as a development methodology. At first, the transition of the knowledge model (as a final product of knowledge acquisition) to software was the poor cousin. Recently, this has been improved considerably through formalization of the KADS knowledge model [Akkermans89]. Nevertheless, issues on testing and maintenance, for instance, are not or just barely addressed in KADS publications. Contrary to [Chandrasekaran87], where generic tasks also play a role, be it that its content differ from the generic tasks. Chandrasekaran emphasizes, as does Breuker c.s., the importance of building bricks for knowledge modelling that possess a substantially higher level of abstraction than matters like production rules, frames and networks. Building on earlier work in the field of diagnosis and design, he presents four generic tasks for diagnosis, hierarchic classification, hypothesis evaluation ('what is the concord between data and hypothesis?'), abductive assembly (to join a number of low-level hypotheses together into a single, coherent unit) and database inference [Chandrasekaran83,84,86]. Besides these four, Chandrasekaran mention two others in relation to design: object-synthesis-through-slection-and-refinement-of-plans and situational-space-abstraction. For all six tasks, generic tools are available. Chandrasekaran explicitly mentions the *undesirability* of uniform knowledge representation for various, generic tasks. He sees intelligence as an interaction of functional units (problem-solvers) that each use the knowledge representation and inference methods according to their type.

The latter statement comes close to the philosophy behind blackboard systems and object-oriented programming. The blackboard approach was not originally developed to improve quality assurance of expert system development, but to tackle complex problems such as speech recognition [Erman80,Fennell77]. As such, a blackboard architecture is eminently suitable for problem-solving where hypotheses that are very uncertain at first are accepted or rejected by another (i.e. by looking at it from a different angle). This approach to problem-solving, however, lends itself to many domains. Physicians, for example, use nosological, epidemological, physiological and anatomical knowledge in the reasoning (e.g. diagnosis). Barristers use jurisprudence alongside the law. Generalizations (of domain-specific blackboard systems) such as AGE/SHELL, Hearsay-III, BB1, MXA and BLOBS are characteristic for the ambitions in the direction of general applicability of blackboard technology [Engelmore88]. Besides the use of more than one inference method and a measure of uncertainty in many domains, there is yet another point that makes the blackboard paradigm an important concept for expert systems. A blackboard architecture offers good possibilities to build in a reasonably advanced modularity and as such, deliver a more orderly product that is easier to test. This aspect has, as yet, received little attention which is probably due to the fact that, until recently, the development of a large blackboard system was next to slavery. Now that more and more blackboard development systems become available, like BB1 [Hayes-Roth88] and GBB [Corkill88], blackboard architectures will probably gain more acceptance. Especially GBB (Generic BlackBoard) is interesting from a quality assurance point of view because of the strong emphasis that is put on specification of the data structure before implementation in the system. A clear parallel is drawn with data definition in database methodology. Furthermore, GBB pays much attention to efficiency in entering of new data and retrieving existing data (blackboard objects in this case). This is an important aspect, given the sorrowful reputation of blackboard systems in this respect. Another generalization that particularly emphasizes the usefulness of blackboard systems in relation to quality assurance is the Edinburgh Prolog Blackboard Shell [Engelmore88]. As a recent example of blackboard technology efforts of national origin, we mention the legal expert system Prolexs [Walker88].

Another trend is the increasing attention for integration of database technology and knowledge engineering. This is exemplified by MEDES, a medical system that lies somewhere in between a database system and a development environment for medical expert systems [DeVriesRobbé89]. The domain expert uses software that is more like a database than an expert system, since all he or she can enter are relationship instances. The queries an end-user can pose, however, make the system an expert system to that end-user. MEDES has three layers. The inner layer is a general purpose shell. Around that we find a layer with inference algorithms and predicate definitions that may (generally) be considered of importance to the medical domain. The outer layer consists of static domain- knowledge. This layer (and in principle this layer only) is filled by a domain-expert, and he or she can only use those predicates that are defined in the middle layer (the so-called subshell). Because inference primitives to be used in queries, are defined in the subshell not accessible for the domain-expert, it is expected from such a system that it allows a rapid, flexible implementation of a new (medical) domain that is not messed up by annoying typing errors. Of course, a predefined set of predicates implies limitations to a domain-expert entering a(nother) medical domain. Experience must demonstrate whether this classifies as an irresolvable hindrance or a welcome limitation. However, it is certain that when one endeavours to promote expert system development form craft to industry, such a limitation of liberties is an appropriate (and possibly the only) way.

5.3.3 Cohesion and coupling in knowledgebases

At least as far as the definitions in [Gane86] and [Yourdon79] go, coupling and cohesion are characteristic of *procedural* software. This does not mean, however, that these cannot relate to knowledgebases: when a knowledgebase is subdivided in modules it is very well possible that a change in one module causes changes in other modules, e.g. in order to keep the knowledgebase consistent as a whole. In fact, [Nguyen87]'s algorithm for consistency control uses a modular subdivision of the knowledgebase to perform a check more efficiently, be it that the subdivision into modules is done *afterwards* (i.e. after the development of the system). Because this is done purely on the basis of dependency between predicates without taking into account the conceptual unity of modules, there is no procedural cohesion involved. It cannot be determined whether the coupling of the modules in the context tree is data or control-coupled. This depends on whether during the inference process, new (derived) facts are being stored.

In blackboard architectures with a single blackboard, i.e. not divided in panels, the *coupling* between the separate knowledgebases is in principle a form of common environment coupling. Usually, blackboards are divided in panels and for every knowledgebase it is specified what panels may be read and which may be written on. In that case, the coupling may be better described as object-oriented. In both cases, this basically involves data-coupling in the sense that a change in one knowledgebase produces no unwanted side-effects in another knowledgebase, at least not if the design of the knowledgebases has not taken the functionality of those other knowledgebases into consideration. An exception to this is changing the access rights (read / write) of a knowledgebase on the various blackboard panels. Assuming that the access rights form a neatly arranged unit, this will not give any problems. Such a change might, however, be considered as a change of the goal specification, something that would not occur in regular maintenance. In a blackboard system such as Hearsay-II, the various knowledgebases are distinct from each other, not only procedurally but also conceptually. Therefore, no *functional* cohesion is involved. Neither is module *hierarchy* in ordinary blackboard systems. Distributed blackboard systems possess a limited (two-layer) module hierarchy [Durfee88].

6 INTEGRITY CONTROL ALGORITHMS

6.1 Introduction

Integrity control in databases has been further developed than the control of consistency and completeness in expert systems. The so-called expert database systems or knowledgebase management systems resemble as far as implementation is concerned, a database, but as far as the outer texture is concerned (as well as deductive capabilities), they look more like an expert system. For some time now, research has been going on with respect to (more) powerful integrity control, as far as expressional power of the constraints and efficiency is concerned.

Each of the paragraphs contained in this chapter comprises the description of an algorithm with which one can check the integrity of a knowledgebase. There are, however, important differences between the algorithms. They can be described as:

- deductive database perspective (Decker);
- programming language perspective (Shapiro);
- production rule perspective (Nguyen).

6.2 Decker's algorithm

The algorithm described in [Decker86] is meant to control the integrity of a deductive database. A deductive database contains three types of data. Beside the facts and constraints that we also find in ordinary databases, it also contains derivation rules. With these rules, other facts can be derived from those explicitly incorporated (the extension of the database). The derivable facts are only implicitly present and are only derived when triggered by a database query. Because of the presence of derivation rules, a deductive database qualifies as a *knowledgebase*. As such it is an alternative for production rule systems. The integrity of a deductive database *is* the integrity of the ordinary database that is obtained by the exhaustive application of derivation rules. Whereas the constraints in an ordinary database limit the set of facts contained (explicitly) in the database, the constraints of a deductive database system relate to the whole set of (either explicit or derivable) facts. In databases as well as deductive databases, the term integrity *check* is used which points out that the main concern is maintaining the integrity when changing the database or knowledgebase, respectively. The algorithms for checking integrity are therefore equipped to efficiently perform an integrity check after a (small) change of the datafile. In the case of databases as well as deductive databases it is assumed that the file complied to the constraints before the update took place. With the help of this assumption, constraints may be simplified in terms of compile time. It is now possible to determine whilst updating, on the basis of contents and type of the update, which of these simplified constraints must be evaluated. How this works will be exemplified at the end of this paragraph.

In deductive databases, queries as well as derivation rules are generally Horn-clauses. Horn-clauses are insufficiently expressive to be constraints. Instead, larger subclasses of first order logic are used such as allowed formulas [Topor86] or range-restricted formulas [Nicolas82,Decker86]. Deductive databases have until now been the subject of theoretical research, using internal Prolog databases. [Decker86] pays much attention to working with range-restricted formulas, especially in checking whether a first-order formula is range-restricted and the description of range-restricted formulas in the canonical form, the so-called range-form. This standard form offers the possibility to write a simple interpreter in Prolog that can evaluate all range-restricted formulas. Decker's discussion about range-restricted formulas is hard to follow for those that do not have received any training in logic and logic programming. For this reason, backgrounds and argumentation receive only a brief discussion. Subjects that do receive indepth discussion are: the definition of range-restricted, the several shapes that range-forms can assume and the simplification algorithm that enables (more) efficient integrity checking. The definition of range-restricted as well as the simplification algorithm are generalizations of an approach to integrity checking in ordinary databases, as is proposed in [Nicolas82].

Definition :

Let F=QM be a formula disjunctive minimum form, thus a disjunction of conjunctions, where Q is the concatenation of quantifications ($\forall x \exists y \forall z$...etc) and M a quantor-free formula.

1. If X is an existentially quantified variable in F (in other words, $\exists$ occurs in Q), then F is range-restricted in relation to X if every conjunction of M with X in it, contains at least 1 positive X-literal.
2. If is X universally quantified in F (in other words, $\forall$ occurs in Q), then F is range-restricted in relation to X if the disjunctive minimum form of $\neg$ F is range-restricted in relation to X.

3. F is range-restricted if F is range-restricted in relation to all variables in Q.

(An X literal is an atom containing X, e.g. p(X) or r(X,Y), or its negation).

The essence of this definition that is probably hard to scrutinize, may become clearer with help of the example below.

Example:

Suppose one wishes to include a constraint in a database saying that the defence staff breaks down into civil personnel (b) and military personnel (m), with no personnel being member of both denominators. First-order logic can formulate this constraint as follows:

$$\forall X((b(X) \vee m(X)) \wedge \neg (b(X) \wedge m(X))) \qquad (1)$$

This formula, however, is not range-restricted. A disjunctive minimum form of it is:

$$F = \forall X(b(X) \wedge \neg m(X) \vee (m(X) \wedge \neg b(X))) \qquad (2)$$

In terms of the definition of range-restricted, this formula relates to case (2) and is F equivalent to:

$$\neg \exists X(b(X) \wedge m(X) \vee \neg b(X) \wedge \neg m(X)) \qquad (3)$$

This formula is not range-restricted because the conjunction $\neg b(X) \wedge \neg m(X)$ does not contain a positive X-literal. Should one attempt to represent this formula in Prolog and encounter:

```
not ( (b(X), m(X)) ; (not b(X), not m(X)) )          (4)
```

then the integrity check in the database below comes to a standstill. Evaluation of (4) yields the result 'yes', but without the specification whether Joe belongs to either civil or military personnel.

```
function(jim,quartermaster).
function(john,technician).
function(joe,chauffeur).
military(jim).
civilian(john).
```

Figure 6.1: Database belonging to the example.

The fact that the integrity check fails is due to the semantics of the not-operator in Prolog, in other words the negation-as-failure rule. The Prolog expression (4) is in fact not the translation of the first-order formula (3), but of:

$$\neg\, [\exists X\, (b(X) \wedge m(X)) \vee (\neg\, \exists X\, b(X) \wedge \neg\, \exists X\, m(X))\,] \qquad (5)$$

To avoid such subtle mistakes in the use of the not-operator in Prolog, one can use the rule of thumb that all variables in the argument of the not-operator have to be instantiated (ground terms) before 'not' is executed. This is exactly what is guaranteed by the notion of range-restricted.

If, by insertion of the predicate p (of personnel) instead of (4):

$$\text{not}\ (p(X), (\,(b(X), m(X))\ ;\ (\text{not}\ b(X), \text{not}\ m(X))\,)\,) \qquad (6)$$

is used, the integrity check will run satisfactory. This Prolog expression corresponds with the range-restricted formula:

$$\forall X(\, p(X) \rightarrow (\, b(X) \wedge \neg\, m(X) \vee m(X) \wedge \neg\, b(X)\,)\,) \qquad (7)$$

where p(X) is called the range-expression.

Especially experienced Prolog programmers will ask themselves why everything is so complicated (using range-restricted formulas) when simplicity lives nextdoor (using ground expressions in Prolog). It is true that the notion of range-restricted can be by-passed and one even does not need Decker's meta-interpreter (to be discussed below) for the evaluation of constraints when using Quintus Prolog. Range-restricted constraints can be included as ordinary Prolog goals, provided that one has *operators* for *and*, *or* and *not* at one's disposal (as, for instance, in Quintus Prolog). Nevertheless, reading and formulating constraints is enhanced when these can be formulated in a predicate logic form. The transition of first-order (range-restricted) expressions to range forms that can be evaluated or even Prolog goals can, in principle, be automated. The notion of range-restricted therefore, brings us closer to declarative programming and as such, can be worthwhile for the quality of software for expert systems.

Decker proves that all range-restricted formulas can be transcribed into the following standard form(s). Either:

1. F only contains existentially quantified variables (with every conjunction containing a Y, contains at least one positive Y-literal), or,
2. F has the form of $\exists X(P \wedge F') \vee F''$, with P being a so-called range-expression in X (a purely existentially quantified disjunction of positive X-literals) and F' and F" range-restricted formulas in standard form, or,
3. F has the form of $\forall X(P \rightarrow F') \wedge F''$, with again P being a range-expression in X and F' and F" range-restricted formulas in standard form.

Furthermore, Decker states that the problems as far as the evaluation of a range form in Prolog are concerned, are limited to version 3 of the range form. If the interpreter can handle range forms formed as $\forall X(P \rightarrow F')$, it can evaluate all range forms.

After all, the existential quantor is implicitly present for all variables in Prolog clauses. By using the auxiliary predicate 'forall' (in the shape of forall(X,Xrex,Concl) as the translation of $\forall X(Xrex \rightarrow Concl)$) and to write a meta-interpreter (see figure 6.2) that can process this predicate and directs all expressions that occur in constraints that do *not* contain forall in the regular Prolog interpreter, the problem can be solved.

For clarity's sake, it must be noted that the first argument for the forall predicate may be omitted here. It has only been added for the sake of readability, not for the sake of evaluation. The algorithm supposes in fact that Xrex is a range-expression in X. Should this not be the case then the constraint will have to be formulated differently.

```
evaluate(Constraint):- verify(Constraint).

verify(forall(X,Xrex,Concl) ):-
   call(Xrex),
   falsify(Concl),!,fail.
verify( forall(X,Xrex,Concl) ):- !.
verify(Constraint):- call(Constraint).
verify(Constraint):- !,fail.

falsify(Constraint):- verify(Constraint),!,fail.
falsify(Constraint).
```

Figure 6.2: Prolog meta-interpreter.

As touched upon before in the discussion of the notion of range-restricted, the meta-interpreter may be forgotten if one uses the Prolog expression 'NOT (Xrex, not Concl)' instead of 'forall (X, Xrex, Concl)'.

The simplification of constraints in updates of *ordinary* databases is quite simple. In the case of an INSERT, true is substituted in the constraints in those places where the fact to be inserted occurs. In case of DELETE, a false is substituted. This needs to be done only once (compile time).

Example:

From the constraint:

$$\begin{aligned}&\forall X(\neg \text{personnel}(X) \vee\\&\qquad(\text{military}(X) \wedge \neg \text{civilian}(X)) \vee\\&\qquad(\text{civilian}(X) \wedge \neg \text{military}(X)))\end{aligned}$$

the following (compile-time) update-conditions are derived for the various kinds of updates:

INSERT personnel(X) ONLY-IF military(X) $\wedge \neg$ civilian(X) $\vee$
civilian(X) $\wedge \neg$ military(X)
INSERT military(X) ONLY-IF $\neg$ personnel(X) $\vee$ not civilian(X)
INSERT civilian(X) ONLY-IF $\neg$ personnel(X) $\vee$ not military(X)
DELETE military(X) ONLY-IF $\neg$ personnel(X) $\vee$ civilian(X)
DELETE civilian(X) ONLY-IF $\neg$ personnel(X) $\vee$ military(X)

When, at this point, an actual update takes place, for instance, INSERT civilian(john), only $\neg$ personnel(john) $\vee \neg$ military(john) has to be evaluated.

While checking the integrity of deductive databases, not only the update itself has to be determined but also its consequences (in terms of derivable facts) in order to locate the constraints that must be evaluated. Actually, two sets of facts must be calculated, i.e. the set of facts that can be derived after the update has taken place and which could not be derived before that, and the set of facts that originally could be derived before the update but cannot afterwards. The sets are indicated with A+ and A-. In this respect it is important to realize that adding a fact to A+ as well as A- can contribute to the presence of 'not' in derivation rules.

The same goes for the deletion of a fact. Decker first describes how the sets of A+ and A- could be calculated straightforwardly, and then presents a (more) efficient procedure for determining B+ and B- as estimators of A+ and A- with A+ $\subset$ B+ en A- $\subset$ B-. Decker relies on the reader's own alertness to note that B+ and B- are not obligatorily equivalent to A+ and A-.

The straightforward method consists of direct application of the definitions af A+ and A-. Let LB be the database before the update and DBN after the update, DB* the extension of DB and DBN* that of DBN. These extensions are defined as the merger of the explicit facts with the implicit ones (derivable form the derivation rules of the database). As Head:-Body is a derivation rule then set_of(Head,Body,Set) yields the set Set of all instances of Head that are a memeber of the extension and can be derived using the appropriate derivation rule. By merging all these sets with explicit facts the complete extension can be obtained. Subsequently, A+ and A- can be calculated from DB* and DBN* via A+ = DBN*\DB* and A- = DB*\DBN*. Of course, calculating A+ and A- in this manner is not very sensible. It is so time-consuming that simplification of the constraints would yield a nett loss of time rather than gaining it. The more efficient method suggsted by Decker comes down to introducing an element of chance and exchanging space complexity against time complexity: of every derivation rule in the database a track is made of which literals can contribute to its conclusion, either in a positive or negative sense. See figure 6.3 for an example.

```
p(X):- q(X),r(X),not s(X).

occurs_positive(q(X), (p(X):- q(X),r(X),not s(X)) ).
occurs_positive(r(X), (p(X):- q(X),r(X),not s(X)) ).
occurs_negative(s(X), (p(X):- q(X),r(X),not s(X)) ).
```

Figure 6.3: Transformation of a Prolog-rule according to Decker.

Now, it is no longer necessary to determine the extension of *all* derivation rules in DB and DBN. *Instead, only the extensions of the derivation rules in the* update will be determined and these give a first contribution to B+ and B- (together with the facts in the update).

Next, the transitive closures of B+ and B- are determined using the meta-information about the rules, as defined in the 'occurs'-facts.

If that is the case, the extension of the corresponding instantiation of the second argument is added to B+ and B- (if a fact from B+ can be unified with a literal in an occurs-negative fact to B- , if a B- fact with an occurs-positive-literal can be unified with B- ... etc.). In fact, after every addition B+ or B-, there is an immediate inspection whether the added element can be unified to a literal in an update constraint. If that is the case, the corresponding (possibly instantiated) condition is checked immediately. This jumping to-and-fro (interleaving) between the expansion of B+ and B- and the actual integrity check increases the chances of fast discovery of a transgression against integrity. After all, the elements added to B+ and B- through the occurs-predicates become more guessable as the number of occurs-facts used increases.

6.3 Shapiro's algorithm

We have already touched upon the fact that Shapiro's algorithm starts with a number of assumptions concerning the formalism on which the debugging-algoritms are applied [Shapiro84]. This formalism, i.e. a program in a certain programming language, can be seen as a reflection of the specification of a problem-domain. Firstly, the basic element of this language is the procedure (or function). During the debugging process only the calls of these procedures are of importance, together with input and output. Secondly, every procedure has a name, arity (the number of arguments) and the corresponding code. Thirdly, the program code registers a set of actions. Fourthly, all calculations of the program, i.e. series of subsequent procedure calls, can be described with a tree-structure that contains the calls in its nodes and points of succes or failure in its leaves.

Shapiro's algorithm was primarily aimed at program debugging; the location and correction of errors in program code [Hamdan89]. These errors can have different causes, such as inadequate functional design or a failing system development method. According to Shapiro, however, a more fundamental cause can be pointed out. He sees a program as the epitomization of a complex set of assumptions. The behaviour of a program is a derivation of these assumptions, which makes it difficult to predict its behaviour completely. The considerations were an incentive for Shapiro to develop a theory that must produce an answer to two questions:

- how can an error be detected in a program that does not function properly?;
- how can this error be corrected?.

This theory has led to the construction of algoritms for each of these two problems. With the help of a diagnosis-algorithm, errors must be detected, after which improvements can be made using the error-correction algorithm. These two algorithms are both included in a so-called 'Model Inference System' (MIS) which takes a program that must be debugged for its input and a list of input / output examples, that partly determines the behaviour of the program.

It is necessary that during the debugging process it is clear what the expected behaviour of the program will be. It is always assumed that there is an authority that can give an answer to the questions that are posed by the system about the 'Universe of Discourse'. The answer can be given by the developer of the program, or by another program. When, for instance, a properly functioning version of a program will be altered, questions that are generated during debugging can be 'answered' by the old version. It is even possible to build a (simple) program from scratch by starting with an 'empty' program. MIS as well as the programs to be debugged are written in Prolog.

The integrity check of knowledgebases is chiefly concerned with detecting errors in 'executable specifications'. It is assumed here that the specification of a knowledge-domain has been established with the help of ENIAM. The resulting conceptual model can be transformed into a formal representation in Prolog that can be executed on the computer. This reprot pays special attention to what is called non-graphic constraints in traditional NIAM. The rules form an important part of the knowledge in a knowledgebase and may be compared to production rules. The diagnostic part of Shapiro's algorithm can check the integrity of this Prolog code.

By applying Shapiro's algorithm to 'pure' Prolog programs, three types of errors can be detected; termination of a program with incorrect results, termination with missing results and non-termination. The basis of the diagnostic algorithm is formed by a meta-interpreter, which is an interpreter for a programming language written in that same language. The meta-interpreter enables the user to subject the calculation process of a program to further scrutiny. As such it may be compared with the trace facility that is used in languages such as C, Pascal, Prolog and Lisp. A meta-interpreter is a meta-program that can treat other programs as data [Sterling86], which simplifies their manipulation, analysis and simulation. At this point, an instructional example of a meta-interpreter for 'pure' Prolog (i.e. simple Prolog without operators such as 'cut' and 'not', for example) is in order:

This meta-interpeter keeps a record in the form of a proof-tree of every proof that must be given. Briefly, its operation is as follows. The atom 'true' is always true. The conjunction of two goals is true if they both are true. A goal containing a body with subgoals is true if all subgoals are true. The last solve/2-line of figure 6.4 contains the predicate 'clause(A,B)'. This separates the head and body of a line; B contains the conjunction of subgoals of a goal named A. Taking as a rule 'A:- B_1, B_2, B_3, B_4.', the second argument of solve/2 assumes the form: (B_1, (B_2, (B_3, B_4))). To be able to debug a program, the developer must have an idea of the program's behaviour when applied to a certain field of application. With the help of a meta-interpreter this program can now be simulated. Using the debugging algorithm, it will then be attempted to detect a difference between the expected and the actual behaviour.

```
solve(true,true).
solve((A,B),(ProofA,ProofB)):-
        solve(A,ProofA),
        solve(B,ProofB).
solve(A,(A:-Proof)):-
        clause(A,B),
        solve(B,Proof).
```

Figure 6.4: Meta-interpreter for simple Prolog.

Before giving a more precise description of Shapiro's algorithms, it might be clarifying to say a few words about the semantics of 'logic programs', Prolog in this case [Shapiro84;Sterling87].

The clause 'A:- $B_1,B_2,...,B_n$.' contains A' in the interpretation M (i.e. *M*eaning) if a substitution θ exists unifying A with A' and makes the goals $B_1,B_2,...,B_n$ in M true. A clause is 'true' in M if every variable-free goal of this clause is contained in M, and 'false' if not. This assertion can easily be reflected in Prolog. In this environment, facts are true when they find themselves in the internal database, or when they can be derived by applying a rule. A program P in M is true if every clause in P is true in M. The notion 'true' may in a logical context be taken as '*correct*', 'false' as '*not correct*'.

The interpretation (meaning) of a program P is $M(P)$, which corresponds with the set of variable-free (fully instantiated) goals in P that are true. Now we can say that a program P is ***complete*** in M if $M \subseteq M(P)$. A program P is only then ***complete*** as well as ***correct*** in M if $M = M(P)$.

The domain D of a program is the set of variable-free goals of P. A program P ***terminates*** on a domain D if there is not a single goal A in D that leads to an infinite derivation of goal A in P.

Previously three types of errors were described that can be detected with the help of Shapiro's algorithm. The basis of this is always the 'proof-tree' that is built during the execution of the program. This may also be represented by a 'top-level trace' of a procedure (predicate in Prolog) p with x as input and y as output. This trace consists of a (possibly empty) set of triplets in the form of $\{ <p_1, x_1, y_1>, <p_2, x_2, y_2>,..., <p_n, x_n, y_n> \}$. If the set is empty, then the procedure p with input *x* can yield the output *y*, without procedure calls. If the set is not empty, then a chain of procedure calls with *x* as its initial input, will eventually yield an output *y*.

A proof-tree is ***complete*** if all nodes are triplets with the form <p, *x*, *y*> and all leaves have the empty set as 'top-level trace'. It can be said that *y* is the ***correct*** output of the procedure call <p, *x*> in ***M***, if <p, *x*, *y*> is contained in ***M***.

A program ***P*** is ***correct*** in ***M*** if the root of every complete 'proof tree' of ***P*** is contained in ***M***. A program ***P*** is ***complete*** in ***M*** if every triplet in ***M*** is the root of a complete 'proof tree' of ***P***. A program ***P*** is ***terminating*** if the 'proof tree' contains no infinite path.

Now we will demonstrate an example of an algorithm for detecting an incorrect solution, based on [Shapiro84;Sterling87]. A program P can only yield an incorrect solution if it contains an incorrect clause. A clause C is not correct within the interpretation ***M*** if [illegible]re is an instantiation of the body of C that is true in ***M*** and a head that is not true in ***M***. The description is as follows:

- Input: procedure p in ***P*** and an input *x*, such that <p, *x*> yields an output *y* that is not correct in ***M***.
- Output: a triplet <q, *u*, *v*> not in ***M***.
- Algorithm: simulate the execution of p on input *x* with *y* as result; if a call <q, *u*> leads to the output *v*, establish (by letting the user pose a question) if <q, *u*, *v*> is contained in ***M***. If this is not the case then yield the incorrect solution <q, *u*, *v*> and stop.

It is hard to determine whether a program will not terminate, i.e. run indefinitely. Shapiro opts for a pragmatic solution to this problem by providing the meta-interpreter a preset limit to the depth of recursion. One of the arguments contains a proof tree of procedure call upto the moment of termination. By examining this trace further, diagnosis of the problem can take place. Beside errors in the program, the cause may also be found in a lack of memory capacity, being responsible for a stack overflow. The description of the algorithm for detecting non-termination is as follows [Shapiro84;Sterling87]:

- Input: procedure p in ***P***, an input *x* and an integer d > 0, such that the call <p, *x*> will not exceed the maximum depth of recursion d.

- Output: when exceeding **d**, *y* will contain the set of procedure calls running the length of **d**, consisting of triplets with the form <**p**, *x*, *y*>.
- *Algorithm: simulate the execution of* **p** *on x; if depth* **d** has been reached without finding a solution, then the program will not terminate.

Of the three types of errors that can be found with Shapiro's algorithm, detecting a missing solution is the hardest one to solve. The notion of 'cover' plays an important role here. A certain clause C is a cover of goal A in relation to interpretation ***M***, if there is an instantiation of C of which the head corresponds with A and the body is contained in ***M***. If a program ***P*** has a missing solution in interpretation ***M***, then there is a goal A in ***M*** that has no cover in a clause of ***P***. During the operation of the algorithm below, the user must answer questions about the validity of certain instantiations.

A program ***P*** is complete in ***M*** if for every triplet <**p**, *x*, *y*> in ***M*** the call <**p**, *x*> yields the output *y*. *This means that* **p** *must be a cover for* <**p**, *x*, *y*>, *if it is not,* then we have of a missing solution. The remedy then, is to change the procedure **p**, such that *y* can be a result. The algorithm for detecting a missing solution is as follows [Shapiro84;Sterling87]:

- Input: procedure **p** in ***P***, an input *x* and an output *y* all contained in ***M***, which make **p** fail.
- Output: a triplet <**q**, *u*, *v*> in ***M***, for which **q** is no cover.
- Algorithm: simulate the execution of **p** on *x* with *y* as a result, using the existential queries and simultaneously keeping track using a proof tree; in case of a 'fail' the triplet <**q**, *u*, *v*> is produced and the algorithm will stop.

6.4 Nguyen's algorithm

Nguyen has developed two algorithms, elaborating on the concepts introduced in [Suwa82], to check the consistency and completeness. The CHECK-algorithm [Nguyen85,Nguyen87b] is closest to the Suwa-approach. Besides contradiction, redundancy and subsumption, under the denominator consistency CHECK also searches for unnecessary conditions and circular rule-chains, for the sake of consistency.

An example of unnecessary conditions. The rules:

IF $p = 1$ AND $q \neq 1$ THEN A en IF $p \neq 1$ AND $q \neq 1$ THEN A

may be merged into:

IF $q \neq 1$ THEN A.

The fact that circular rule-chains are checked already indicates that CHECK not only looks at flat (parts of) rule-files. This is also reflected in the checking of completeness. Beside conditions that are not covered by parameter values, a search is also made for unreachable conditions, dead-end conditions and dead-end goals.

The CHECK-algorithm is suitable for diagnosing knowledgebases made with the help of a '*generic expert system shell*' (*Lockheed* Expert System [Laffey86] in this case). The rules in LES assume the form IF<conjunction of conditions>THEN<conjunction of conclusions>. Beside backward-chaining, LES also provides for forward-chaining. The discussion here is limited to backward-chaining, because there is not much difference in integrity-checking in forward and backward-chaining. Only the notion of unreachable conclusions is meaningless in forward-chaining.

Like ONCOCIN, LES uses a kind of context-mechanism. Anyhow, Nguyen only applies an integrity-check to component *parts* of the whole rule-file of an application, thus substantially reducing the computational costs of the algorithm. Furthermore, LES-rules can contain variables.

[Nguyen85] gives a description of the algorithm in an Algol-like notation. However, this is so incomplete that it is not clear how a consistency-check is performed, let alone the possibility of making a statement about its efficiency: declarations of variables are not included and the result-type of the function Compare_clauses (..., ...) must be guessed. The function itself has not been worked out. The same goes for the Transform function in the allocation 'matched_rule = Transform (i, k, clause-relations)'.

[Nguyen87b] verbally (i.e. not Algol-like) describes how the algorithm basically works. Of course, things are associated with the working method of LES. The main issues are as follows.

Nguyen remarks that in LES, multiple goals can be 'active', with one or more rule-sets being coupled to each goal. A rule-set contains production rules that can contribute to accomplishing the goals involved. Nguyen also notes that the rule-sets of the goals may be tested separately and it does not form a significant limitation for the type of production rule systems for which the algorithm can be used. In case of a possible unwanted interaction or dependability between the rules in different sets, these sets are merged into a new, larger set. Please consider the following. A rule-set consists of a number of rules. Every rule contains an IF-part and a THEN-part. The IF-part contains one or more conditions, the THEN-part one or more conclusions.

It is not clear what the individual conclusions and conditions exactly look like (assuming that 'IF ?X has a fever, AND ?X has flat pink spots on his skin THEN type of disease of ?X is measles') is a 'natural language' representation of 'IF symptom(?X, fever) AND symptom(?X, flat_pink_spots_on_skin) THEN disease_type (?X, measles)' *or something similar*. In the latter form, the IF-part thus contains two conditions, i.e. 'symptom (?X, fever)' and 'symptom (?X, flat-pink- spots-on-skin)'. Nguyen [1987b] now sketches the following procedure: First, all individual conditions and conclusions are compared (in pairs). The result is a table with for every pair n of the titles SAME, DIFFERENT, CONFLICT, SUBSET or SUPERSET. From these clause-clause relationships (Nguyen speaks about conditions/conclusions and about IF-clauses/THEN-clauses, interchangeably) the relationships between the IF- and THEN-parts of the various rules are produced. Finally, these are used to indicate whether pairs of rules are conflicting, subsuming, redundant etc.

Beside the CHECK-algorithm, Nguyen also describes the ARC-algorithm. ARC stands for ART Rule Checker [Nguyen87a]. According to Nguyen, this offers additional possibilities in the form of detecting redundant rule-*chains*, subsumed rule-*chains* and conflicting rule-*chains*. Nguyen does not describe how these additions are realized. As said before (in other terms), the three articles that were studied [Nguyen85, Nguyen87a, Nguyen87b], would very well have lend themselves for a 'text-checker' in the spirit of Nguyen himself. They contain a lot of inconsistencies and, what is more important, are very incomplete. On the basis of the information given above, the description of Puuronen's algorithm and one's own ideas, it is probably far more effective to build a rule-checker from scratch than to attempt to reconstruct Nguyen's algorithm. Therefore, let us conclude this discussion about consistency and completeness, the Nguyen way, with stating our *surmise* about the limited possibilities for checking that are offered by the algorithms, as well as our grounds for this surmise.

Surmise: ARC only detects flagrant errors. The assertion that ARC is able to detect conflicting and redundant rule-chains, is misleading. From [Nguyen87a] it can be deduced that ARC can only do this when the first rules of the two chains have the same or equivalent IF-parts. Two IF-parts are equivalent if they contain the same number of conditions and these conditions are equivalent pairwise. Two conditions are equivalent if they can be unified. All this implies that the conflict in the rules below cannot be detected by ARC:

Example:
IF A=1 THEN B=1
    IF A=1 and C=1 THEN D=1
    IF B=1 and C=1 THEN D=0

The same goes for subsumption. And what holds true for ARC, holds especially true for CHECK.

The discussion above does not mean that CHECK (or ARC) is a botch. On the one hand an algorithm that performs a more thorough check is highly desirable from a quality assurance point of view, on the other hand we must not forget the price tag (in terms of complexity).

6.5 A comparative analysis

In the category of production-rule-oriented methods, Nguyen's algorithms perform the most extensive integrity checks. He goes beyond checking one-step inference and also looks at the well-formedness of rule-*chains*. It must be said, though, that the general character of this rule-chain check is somewhat disappointing. Redundant and conflicting rule-chains are, for instance, only detected when the first rules of both chains contain identical or equivalent IF-parts. This type of redundancy c.q. conflict is the most flagrant and thus will be the easiest one to prevent by programming more carefully. Nguyen's consistency check too, does not qualify as being complete.

When you compare the concepts of consistency and completeness with the concept of integrity of deductive databases, then subsumption appears to be out of the question: both explanations of integrity contain elements not mutually shared. Of the various facets of production-rule consistency, the database-oriented approach covers only one (conflict), but this happens in a much wider sense because the latter approach enables the knowledge engineer to impose semantic restrictions to the database in a general sense. Furthermore, it is more thorough. Every conflict that can be derived will actually be detected. This makes the fact that subsumption and redundancy are not detectable less serious. After all, they are not imminently disastrous for the reliability of the expert system. After maintenance (changes in the knowledgebase) they can be a source of contradictions, but as soon as an intended update threatens to cause this, this will most certainly be detected during an integrity check. The absence of a circularity check is more serious. In a certain (ironic) way, circularity is detected because the integrity check comes to a standstill. The complete lack of a check for completeness in the deductive approach too, is a fundamental flaw in this method.

Shapiro's algorithm is designed to detect three possible errors in programs: non-termination, incorrect solutions and missing solutions. Considering the descriptions of consistency and completeness it can be said that Shapiro's *interpretation of these notions* covers various aspects. As far as consistency is concerned, the conflict and circularity are aspects analysed with the help of algorithms for the detection of incorrect solutions and non-termination. The algorithm for detecting a missing solution indicates where there is a missing line in the program that should generate this solution; this covers the completeness aspect.

Non-termination is detected by Shapiro when a preset maximum depth of recursion is reached by the program, without having reached a solution. When compared to the other algorithms, discussed in chapter 3, this approach is very pragmatic, but very effective. In this case, conflict has a slightly different definition. Shapiro's use of the term correctness lends a more formal principle to this problem. The result of the call for a procedure p in an interpretation is correct, if all subgoals of p, either direct or indirect, yield a result that is correct in *M*. Shapiro's algorithm yields in this case a triplet with the form <p, *x*, *y*>, that is the cause of the incorrect result. If a missing solution is the subject, then a distinction must be made between deterministic (e.g. Algol-derivatives and functional languages such as Lisp) and non-deterministic languages (e.g. logic programming languages such as Prolog). Suppose that *y* is the correct output of the call <p, *x*>. If <p, *x*> terminates and yields a different value than *y* as well, then this output is, in case of a deterministic language, ***not correct***. In a non-deterministic language it is so that if every call <p, *x*> yields an output that is correct in *M*, but not a single call yields *y*, then the call <p, *x*, *y*> fails. In this case, the ***procedure*** p is ***not complete***. A ***program P*** is complete in *M* for every triplet <p,

x, *y*> in ***M***, if there is a call <p, *x*> that has *y* as for its result. If the program P fails on any triplet <p, *x*, *y*> in ***M***, then ***P*** is not complete in ***M***. The problem can be solved by adapting procedure p, such that *y* will indeed be produced.

It is not very well possible to label one of the types of integrity checks as an all-purpose winner. What is the best method will probably depend on the available running time and memory space. Also the question how large a knowledgebase may become before the calculation time needed for an integrity check becomes unacceptable remains unanswered, because the various algorithms could not be tested on realistic applications. However, an answer -reasonably well-founded- can be given to the question which algorithms must first be taken into account for further investigation and which *combination* is expected to give a quality result.

The deductive database approach is especially suited for second generation expert systems. The internal models of such a system in that case form a *fixed* (not changing per session) bank of facts. Even when one puts aside a production-rule formalism, a deductive database offers good possibilities (thanks to Prolog's great flexibility and the fact that the integrity check uses, in the last place, the Prolog interpreter). Of the two methods discussed in this category, Decker's method is by far the easiest to implement. This method could be enhanced with a Shapiro-like circularity detector and possibly the completeness check like that of Nguyen. Furthermore, it is absolutely necessary that, in the case of 'user askable' parameters, the integrity check contains all possible (permitted) combinations of parameter values.

Shapiro's algorithm offers good possibilities for the integration of conceptual modelling (ENIAM) and integrity checking. In determining non-termination, it is felt that the algorithm lacks more support in further diagnosing the proof tree that is produced. Adding other techniques as is suggested by Nguyen is worth considering. From the analysis of correctness and completeness it has become evident that the user plays an important (interactive) role. By selecting variants with the most favourable query complexity and extending the algorithms to 'full-fledged' Prolog, an integrated AI development environment can be realized. The possibilities offered by the Model Inference System, including automatic repair of incorrect procedures, can certainly play a role in this.

7 CONCLUSIONS

The quality framework as presented at the beginning of this study, forms an adequate guideline for the approach of quality problems in expert systems. In concrete terms, this means that the notion of quality will be looked at from three angles: the system specifications, the structuring of the development process and the expert system as a product. Further elaboration of validation and verification methods and techniques for specifications as well as the final product have been reviewed. Furthermore, guidelines have been given for a structured development process of expert systems that can be properly controlled.

As the most important part of an expert system, the knowledgebase has been mentioned. Reasoning mechanism, man-machine interface and the explanation facility contain so many conventional components that quality assurance can take place with many methods and techniques that are already widely used.

Reliability and maintainability are the most important criteria associated with expert systems development.

The solution of the aforementioned problems encountered in practice, must in the first place be searched in improving the modularity and integrity of knowledgebases.

Two technologies from the realm of software engineering seem to occupy the best slot to help realize these improvements. Especially database technology is able to contribute the necessary concepts and methods for controlling integrity. It is true that not all integrity constraints necessary for knowledgebases can be traced back to conventional database systems, but the methods and techniques designed for checking integrity in *deductive* database systems are so wide that integrity constraints that fall under the denominator of consistency and completeness are covered. Furthermore, the use of database technology offers additional advantages as far as security, recovery, concurrency and distribution are concerned.

For the application of a modular structure, database technology is somewhat less suitable. Blackboard technology seems in this respect to be a better alternative.

The development process of expert systems is difficult to control. Contrary to phasing methods for conventional system development, there is no such approach for expert systems that is generally accepted. The experience that is gained over the last few years in using methods like SKE and a phasing as in the Weitzel-Kerschberg life cycle model, will in the future possibly offer some alleviation. Quality assurance is only possible if a feasibility study is made in the first phase, just as it is done in conventional system development.

As yet, there is no suitable (and tested) method that can be used in making clear specifications of expert systems. The development of KADS qualifies as an incentive, but the types of expert systems that lend themselves for the application of this method, remain limited. It is possible that this problem can be solved to a certain extent by using an extended conventional method, such as ENIAM. This will also involve the necessary attention to the way in which knowledge will be elicited from the expert.

E(xtended)NIAM offers better possibilities to record the specifications of a knowledgebase than NIAM does. The possibilities to actually represent non-graphic constraints of NIAM in a graphic way, in particular, is of great importance to the expert for directly validating the conceptual model. It is unfortunate, however, that the other shortcoming of NIAM has not been dealt with yet in the E(xtended) version, namely the explicit registration of temporal aspects.

Evaluating and testing expert systems is as yet an underdeveloped field. Often, it is impossible to test whether the advice given by an expert system complies with an objective standard. For the purpose of user acceptance it is of great importance to know in how far a system meets the requirements and thus forms a quality aspect. Test cases will have to be designed but a structured approach to do this has as yet not been developed for expert systems. Application of traditional testing methodologies may offer a solution in this respect.

The deductive database approach pays too little attention (actually none at all) to circularity and incompleteness when compared with the approaches of Nguyen and Shapiro. This is balanced by the fact that semantic (deductive) database integrity comprises much more than Nguyen's contradictions. Furthermore, an algorithm like that of Decker can be easily implemented in Prolog. By simplifying constraints into update constraints and especially using the meta-information about derivation rules, computational complexity will be substantially reduced. The price that must be paid is the multiplication of the necessary memory space: an average of k literals per rule-body needs about k+1 times of memory space to store the meta-information (occurs_in predicates) together with the original database and its copy.

The aim of Shapiro's algorithm was to design a formal theory for the debugging process. The use of Prolog was instigated by the circumstance that there is a direct relationship between the syntax (structure) and the semantics (meaning) of Prolog. This makes it simple to diagnose and correct faulty programs. This characteristic is very welcome in the analysis of knowledgebase specifications, because a Prolog program may be seen as a formalization of these specifications. So it is more than just a program. However, it must be noted that this study has only dealt with 'pure' Prolog. An extension in the direction of 'full-fledged' Prolog is very well feasible, according to Shapiro [Shapiro84].

By using Shapiro's algorithm it is possible to investigate a knowledgebase specification that can be executed in Prolog, as far as consistency and completeness are concerned. However, it will be necessary that the conceptual model made with the help of ENIAM can be transposed into a Prolog program.

The combination of ENIAM, Prolog and the algorithms of Shapiro and Decker enables the developer of knowledgebase specifications to work on a consistent and complete conceptual model of a problem domain within the same theoretical framework (incrementally). The fact that the method (ENIAM) acts as a starting point, makes it a sturdier approach than that of Nguyen. It is true that the latter performs a very extensive analysis of a knowledgebase, but (as yet) is not interested in the underlying (conceptual) model of reality.

Although these technologies will in most cases show a significant improvement in quality, they are not capable of solving *all* quality problems in expert system software. Knowledge modelling in particular, will always have be a problem for certain domains and as such they are labelled pathologic. This refers to so-called pre-paradigmatic knowledge domains; somehow, these domains have not (yet) been provided with explicit theory in an orderly way. As a practical example a medical domain comes to mind, with concepts that often are not sharply defined and a myriad of synonyms and near-synonyms further obscuring a clear view. In such a domain, the knowledge engineer faces the task of building a coherent and compact theory. Needless to say that for such an *essentially creative* task no ready-made answers may be expected -not from any technology-, other than a certain amount of support. Neither database nor blackboard technology can offer a lot of support in this respect. In some cases the translation of concepts from another domain can possibly offer a solution.

[illegible]

Jacques H.J. Lenting (author)

[illegible]

Michael Perre (author)

[illegible]

Johan Bruin (project leader)

REFERENCES

[ACM82] ACM computing surveys, Issue on Software Validation, Vol. 14, Nr. 2, 1982.

[Addis87] Addis, T.R., "Designing Knowledge-Based systems", 1987.

[Akkermans89] Akkermans, J.M. (et al.), "Naar een Formele Specificatie van Kennismodellen", In: "Proceedings NAIC '89", 1989, pp. 105-119.

[Atzeni88] Atzeni, P.en D.S. Parker Jr., "Formal Properties of Net-based Knowledge Representation Schemes", In: "Data & Knowledge Engineering", Vol. 3, 1988.

[Bakker87] Bakker, R.R., "Knowledge Graphs: Representation and Structuring of Scientific Knowledge", University of Twente, 1987.

[Barachini87] Barachini, F., and Adlassnig, K., "Consded: Medical Knowledge Base Consistency Checking", In: Proceedings Medical Informatics Europe 87, pp. 974-980, Rome, 1987.

[Bennet85] Bennet J.S., "ROGET: A Knowledge-Based System for Acquiring the Conceptual Structure of a Diagnostic Expert System", In: Journal of Automated Reasoning, Vol. 1, 1985, pp. 49-74.

[Berg82] Berg, H.K. (et al.), "Formal methods of program verification and specification", 1982.

[Bezem87] Bezem, M., "Consistency of Rule-Based Expert Systems", Report CS- R8736, CWI, Computer Science/Department of Software Technology, 1987.

[Billault88] Billault, J.P., "Knowledge Acquisition for a Configuration Task", Internal Report, University of Amsterdam, Department of Social Sciences, Amsterdam, 1988.

[Bobrow86] Bobrow, D.G.(et al.), "Expert systems: perils and promises", In: "Communications of the ACM", Sept. 1986.

[Bowen79] Bowen, J.B., "A survey of standards and proposed metrics for software quality testing", In: Computer, Vol. 12, No. 8, 1979, pp. 37-42.

[Breuker87] Breuker, J. (ed.), "Model-Driven Knowledge Acquisition: Interpretation Models", University of Amsterdam, 1987.

[Breuker88a] Breuker, J., Wielinga, B., "Models of expertise in Knowledge Acquisition", In: Guida, G., Tass, C, (eds.), "Topics in Expert System Design", North Holland, 1988.

[Breuker88b] Breuker, J., Wielinga, B., "Kads: een overzicht van een methodologie voor het bouwen van expertsystemen", In: "Proceedings NAIC '88", Amsterdam, 1988.

[Brodie89] Brodie, M.L. (et al.),"Future Artificial Intelligence Requirements for Intelligent Database Systems", In: Kerschberg, L. (ed.), "Expert Database Systems, Proceedings from the Second International Conference", 1989, pp. 45-62.

[Bruin88] Bruin, J.(et al.), "Damocles, een Relationeel Expertsysteem", In: "Proceedings AI Toepassingen '88, 1988, pp. 217-219.

[Buchanan84] Buchanan, B.G. (et al.), "Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project", 1984.

[Buck81] Buck, F.O., "Indicators of quality inspections", IBM technical report TR 21.802, Systems communications division, 1981.

[Buckley79] Buckley, F., "A standard for software quality assurance plans, In: "Computer", Vol. 12, Nr. 8, 1979, pp. 43-48.

[Butler87] Butler, K.A., "Application of Correlation Measures for Vaiidating Structured Selectors", In: "Proceedings IEEE 3rd Conference on Artificial Intelligence", 1987, pp 327-330.

[Bylander86] Bylander,T., Mittal S., "CRSL: A Language for Classificatory Problem Solving and Uncertainty Handling", In: "AI Magazine", Vol. 7, 1986.

[Chandrasekaran83] Chandrasekaran, B., "Towards a Taxonomy of Problem Solving Types", In: "AI Magazine", Vol. 4, Nr. 1, 1983.

[Chandrasekaran84] Chandrasekaran, B., "Expert Systems: Matching Techniques to Tasks", In: Reitman W. (ed.), "Artificial Intelligence Applications for Business", Ablex, 1984.

[Chandrasekaran86] Chandrasekaran, B., "Generic Tasks in Knowledge-Based Reasoning: High Level Building Blocks for Expert System Design", In: "IEEE Expert" ,Vol. 1, Nr. 3, 1986, pp. 23-30.

[Chandrasekaran87] Chandrasekaran B., "Towards a Functional Architecture for Intelligence Based on Generic Information Processing Tasks", In: "Proceedings IJCAI", 1987, pp. 1183-1192.

[Chow85] Chow, T.S. (et al.), "Sof.ware Quality Assurance: A Practical Approach", 1985.

[Clancey81] Clancey, W.J., Letsinger R., "NEOMYCIN: Reconfiguring a Rule-Based Expert System for Application to Teaching", In: "Proceedings IJCAI '81", 1981, pp 829-836.

[Clancey83] Clancey, W.J., "The Epistemology of a Rule-Based Expert System", In: "Artificial Intelligence", Vol. 20, 1983 , pp 215-251.

[Clancey85] Clancey, W.J., "Heuristic Classification", %%[PrinterError: out of paper]%% %%[PrinterE
pp. 289-350.

[Conklin87] Conklin, J., "Hypertext: A survey and Introduction", In: "IEEE Computer", Vol. 20, Nr 9, 1987.

[Corkill86] Corkill, D.D., Gallagher, K.Q., Murray, K.E., "GBB: A Generic Blackboard Development System", In: "Proceedings AAAI", 1986, pp. 1008-1014.

[Creasy88] Creasy, P.N., "Extending Graphical Conceptual Schema Languages", University of Queensland, 1988.

[Creasy89] Creasy, P.N., "ENIAM: A More Complete Conceptual Schema Language", In: "Proceedings of the Fifteenth International Conference on Very Large Databases", 1989, pp. 107-114.

[Cullen88] Cullen, J., Bryman, A., "The Knowledge Acquistion Bottleneck: Time for Reassessment?", In: Expert Systems, Vol. 5, Nr. 3, 1988.

[Curtis79] Curtis, B. (et al.), "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics", In: "IEEE Transactions on Software Engineering", SE-5, 2, 1979.

[Date81] Date, C.J., "An introduction to database systems: Volume I", Addison-Wesley, 1981.

[Date83] Date, C.J., "An introduction to database systems: Volume II", Addison-Wesley, 1983.

[Davis79] Davis, R., "Interactive transfer of expertise", In: "Artificial Intelligence", Vol. 12, 1979, pp. 121-158.

[Davis82] Davis, R., D. Lenat, "Knowledge-based Systems in Artificial Intelligence", McGraw-Hill, 1982.

[Decker88] Decker, H., "Integrity Enforcement on Deductive Databases", In: Kerschberg, L (ed.), "Expert Database Systems, Proceedings of the First International Conference on Expert Database Systems", 1988, pp. 381-395.

[DeGreef88] De Greef, P., Schreiber, G., Wielemaker, J., "StatCons: Een Case-Study in Gestrucureerde Kennisacquisitie", In: "Proceedings NAIC '88", 1988.

[DeMarco85] De Marco, T., "Controlling Software Projects: Management, Measurement and Estimation", 1985.

[Deutsch82] Deutsch, M.S., "Software verification and validation: Realistic project approaches", 1982.

[DeVriesRobbé89] De Vries Robbé P.F., Zandstra P.E., Beckers W.P.A., "Verschillende redeneermechanismen in MEDES", In: "Proceedings NAIC-89", Academic Service, 1989, pp. 59-68.

[Dobbins82] Dobbins, J.A. (et al.), "Software Quality Assurance: Concepts", In: "Journal of Defense Systems Acquisition Management", Vol. 5, Nr. 4, 1982, 108-118.

[Doyle88] Doyle, J., "Expert Systems and the 'Myth' of Symbolic Reasoning", In: "IEEE Transactions on Software Engineering", Vol. SE-11, Nr. 11, November 1988.

[Dunn82] Dunn, R. (et al.), "Quality Assurance for Computer Software", 1982.

[Durfee88] Durfee, E.H., "Coordination of Distributed Problem Solvers", Kluwer Academic Publishers, 1988.

[Elspas72] Elspas, B., "An Assessment of Techniques for Proving Program Correctness", In: "ACM Computing Surveys", Vol. 4, 1972, pp. 97-147.

[Engelmore88] Engelmore, R.S., Morgan A.J., "Blackboard Systems", Addison-Wesley, 1988.

[Erman80] Erman, L.D., Hayes-Roth F.A., Lesser V.R., Raj Reddy D., "The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty", In: "Computing Surveys", Vol. 12, Nr. 2, 1980, pp. 213-253.

[Eshelman87] Eshelman L., Ehret D., McDermott J., Tan M., "MOLE: A Tenacious Knowledge Acquisition Tool", In: "International Journal of Man Machine Studies", Vol. 26, 1987.

[FEL1989-148] Dekker, S.T., Lenting J.H.J., Perre M., Rutten J.J.C.R., "Kwaliteit van Expertsystemen: Een Oriëntatie", FEL 1989-148, 1989.

[FEL-89-A267] Lenting, J.H.J., M. Perre, "Kwaliteit van Expertsystemen: Methoden en Technieken", FEL-89-A267, 1989.

[FEL-89-A312] Lenting, J.H.J., M. Perre, "Kwaliteit van Expertsystemen: Algoritmen voor Integriteits-controle", FEL-89-A312, 1990.

[FEL-90-A012] Lenting, J.H.J., M. Perre, "QUEST: Kwaliteit van Expertsystemen", FEL-90-A012, 1990.

[Fennell77] Fennell, R.D., Lesser V.R., "Parallelism in Artificial Intelligence Problem Solving: A Case Study of Hearsay-II", In: "IEEE Transactions on Computers", 1977, pp. 98-111.

[Forgy82] Forgy C.L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", In: "Artificial Intelligence", Vol. 19, 1982, 17-37.

[Frost86] Frost, D., "Introduction to Knowledge Based Systems", Collins, 1986.

[Gane79] Gane, C., T. Sarson, "Structured Systems Analysis", Prentice-Hall, 1979.

[Gibson89] Gibson, V.R., Senn, J.A., "System Structure and Software Maintenance Performance", In: "Communications of the ACM", Vol. 32, Nr. 3, 1989.

[Gilb76] Gilb, T., "Software metrics", 1976.

[Gissi87] Gissi, P., "Logical Checks in a Rule Based Medical Decision Support System", In: "Proceedings Medical Informatics Europe '87", Rome, 1987, pp 981-985.

[Halstead77] Halstead, M.H., "Elements of software science", 1977.

[Hamdan89] Hamdan, A.R., C.J. Hinde, "Fault Detection Algorithm for Logic Programs", In: "Knowledge-Based Systems", Vol. 2, Nr. 3, 1989.

[Hansen84] Hansen, H.L., "Software validation", 1984.

[Hantler76] Hantler, S.L. (et al.), "An Introduction to Proving the Correctness of Programs", In: "ACM computing surveys", Vol. 8, 1976, pp. 331-354.

[Hayes-Roth83] Hayes-Roth, F., Waterman, D.A., Lenat, D.B., "Building Expert Systems", Addison-Wesley, 1983.

[Hayes-Roth88] Hayes-Roth B., Hewett M., "BB1: An Implementation of the Blackboard Control Architecture", In: Engelmore R.S., Morgan A.J. (eds.), "Blackboard Systems", Addison-Wesley, 1988.

[Hofland87] Hofland, A.G., "The HYDRA Systems" (in Dutch), Internal Report Department of Mathematics and Informatics, Delft University of Technology, 1987.

[Howden82] Howden, W.E., "Life-cycle software validation", In: "Computer", Vol. 15, Nr. 2, 1982, pp. 71-79.

[Hsiao85] Hsiao, J., "Tools for Knowledge Base Management", In: "Proceedings of the Fifth International Conference on Intelligent Systems and Machines", 1985.

[Iman85] Iman, R.L., and Conover, W.J., "A Measure of Top-Down Correlation", Technical Report SAND 85-601, Sandia National Laboratories, Albuquerque, 1985.

[Kerschberg86] Kerschberg, L. (ed.), "Expert database systems, Proceedings from the First International Workshop", The Benjamin/Cummings Publishing Company Inc., 1986.

[Kerschberg87] Kerschberg, L. (ed.), "Expert database systems, Proceedings of the First International Conference on Expert database systems", University of South Carolina, 1987.

[Kerschberg89] Kerschberg, L. (ed.), "Expert database systems, Proceedings of the Second International Conference on Expert database systems", University of South Carolina, 1989

[Kolence86] Kolence, K.W., "An Introduction to Software Physics", 1986.

[Kowalski75] Kowalski R.A., "A Proof Procedure Using Connection Graphs", In: "Journal of the ACM", Vol. 22, Nr. 4, 1975, pp. 572-595.

[Kuhn62] Kuhn, T.S., "The Structure of Scientific Revolutions", University of Chicago, 1962.

[Laffey86] Laffey T.J., Perkins W.A., Nguyen T.A., "Reasoning about Fault Diagnosis with LES", In: "IEEE Expert", Vol. 1, Nr. 1, 1986, pp. 13-20.

[Lenting88] Lenting, J.H.J., "Qualitative Reasoning in Medical Expert Systems", Internal Report MIDS/K88-054, University Hospital Groningen, 1988.

[Liebowitz86] Liebowitz, J., "Useful Approach for Evaluating Expert Systems", In: Expert Systems, Vol. 3, Nr. 2, 1986, pp. 86-96.

[Llinas87] Llinas, J., Rizzi, S., "The Test and Evaluation Process for Knowledge Based Systems", Technical Report F30602-85-C-0313, Calspan Corporation, June, 1987.

[Lloyd85] Lloyd J.W., Topor R.W., "A Basis for Deductive Database Systems", In: " Journal of Logic Programming", Vol. 2, Nr. 2, 1985, pp. 93-109.

[Lloyd86a] Lloyd J.W., Sonenberg E.A., Topor, R.W., "Integrity Constraint Checking in Stratified Databases," Technical Report 86/5, Department of Computing Science, University of Melbourne, 1986.

[Lloyd86b] Lloyd J.W., Topor R.W., "A Basis for Deductive Database Systems II", In: "Journal of Logic Programming", Vol. 3, Nr. 1, 1986, pp. 55-67.

[Marcus87] Marcus, S., "Taking Backtracking with a Grain of SALT", In: "International Journal of Man Machine Studies", Vol. 26, 1987, pp. 383-398.

[Marek87] Marek, W., "Completeness and Consistency in Knowledge Base Systems", In:Kerschberg L. (ed.), "Expert Database Systems", Benjamin/Cummings, 1987.

[Miller78] Miller, E.F. (et al.), "Tutorial: Software Testing & Validation", IEEE Catalog EHO 138-8, 1978.

[Musen87] Musen, M., Fagan, L.M., Combs, D.M., Shortliffe, E.H., "Using a Domain Model to Drive an Interactive Knowledge Editing Tool", In: "International Journal of Man Machine Studies", Vol. 26, 1987, pp. 105-121.

[Myers79] Myers, G.J., "The Art of Software Testing", Wiley, 1979.

[Napheys89] Napheys, B., D. Herkimer, "A Look at Loosely-Coupled Prolog Database Systems",In: Kerschberg, L. (ed.), "Expert Database Systems, Proceedings from the Second International Conference", 1989 , pp. 257-271.

[Nau87] Nau D.S., Reggia J.A., "Relationships between Deductive and Abductive Inference in Knowledge-Based Diagnostic Problem Solving". In: L. Kerschberg (ed.), "Proceedings of the 1st International Workshop on Expert Database Systems", 1987.

[Newell82] Newell, A., "Knowledge Level", In: "Artificial Intelligence", Vol. 18, 1982.

[NGI88] NGI, Proceedings van de conferentie Semantiek in gegevensmodellen, 1988.

[Nguyen85] Nguyen, T.A., Perkins, W.A., Laffey, T.J., Pecora, D., "Checking an Expert Systems Knowledge Base for Inconsistency and Completeness", In: "Proceedings of the Ninth International Joint Conference on AI", Vol. 1, 1985, pp. 375-378.

[Nguyen87a] Nguyen, T.A., "Verifying Consistency of Production Systems", In: "Proceedings of the IEEE 3rd Conference on AI", 1987, pp. 4-8 .

[Nguyen87b] Nguyen, T.A., Perkins W.A., Laffey T.J., Pecora D., "Knowledge Base Verification", In: "AI Magazine", 1987, pp. 69-75.

[Nicolas82] Nicolas, J.M., "Logic for Improving Integrity Checking in Relational Data Bases", In: "Acta Informatica", Vol. 18, 1982, pp. 227-253.

[Nijssen86] Nijssen, G.M., "Knowledge Engineering, Conceptual Schemas, SQL and Expert Systems: A Unifying Point of View", In: "Relationele Database Software, 5e Generatie Expertsystemen en Informatie-analyse", Congres-syllabus NOVI, 1986, pp. 1-38.

[Parsaye88] Parsaye, K., "Acquiring and Verifying Knowledge Automatically", In: "AI Expert", Vol. 3, Nr. 5, 1988.

[Puuronen87] Puuronen S., "A tabular Rule-Checking Method", In: "Proceedings 7th International Workshop on Expert Systems and their Applications", 1987, pp. 257-266.

[Quintus89] Quintus Database Interface User's Guide, Quintus Computer Systems Inc., 1989.

[Quirk86] Quirk, W.J., "Verification and validation of real-time software", 1986.

[Reboh81] Reboh, R., "Knowledge Engineering Techniques & Tools in the Prospector Environment", SRI Technical Note 243, Stanford research Institute, 1981.

[Remus79] Remus, H. (et al.), "Prediction and Management of Program Quality", In: "Proceedings of the Fourth International Conference on Software Engineering", 1979, pp. 341-350.

[Rich79] Rich, C., "Initial report on the LISP Programmer's Apprentice, In: "IEEE Transactions on Software Engineering", Vol. 4, Nr. 6, 1979, pp. 342-376.

[RichardsAdrion82] Richards Adrion, W., "Validation, verification and testing of computer software", In: "ACM Computing surveys", Vol. 14, Nr. 2, 1982, pp. 160-192.

[Robinson65] Robinson J.A., "Automatic Deduction with Hyper-resolution", In: "International Journal of Computer Mathematics", Vol. 1, 1965, pp. 227-234.

[Rousset88] Rousset, M.C., "On the Consistency of Knowledge Bases: The COVADIS System", In: "Proceedings ECCAI '88", 1988, pp. 79-84.

[Sadri88] Sadri, F., Kowalski R., "A Theorem-Proving Approach to Database Integrity", In: Minker J. (ed.), "Deductive Databases and Logic Programming", Morgan Kaufmann, 1988.

[Schneider84] Schneider, M.L., "Ergonomic Considerations in the Design of Command Languages", In: Vassiliou, Y. (ed.), "Human Factors and Interactive Computer Systems: Proceedings of the NYU Symposium on User Interfaces", Ablex, 1984.

[Schreiber87] Schreiber, G. (ed.), "Towards a Design Methodology for KBS", University of Amsterdam, 1987.

[Shapiro84] Shapiro, E.Y., "Algorithmic Program Debugging", The MIT Press, 1984.

[Shortliffe75] Shortliffe, E.H., D. Davis, "SIGART Newsletter", Vol. 55, 1975.

[Shortliffe76] Shortliffe, E.H., "Computer-based Medical Consultations: MYCIN", 1976.

[SKE88] Structured Knowledge Engineering, Syllabus Bolesian Systems Europe B.V., 1988.

[Soper86] Soper P.J.R., "Integrity Checking in Deductive Databases", MSc. Thesis, Department of Computing, Imperial College, University of London, 1986.

[Sowa84] Sowa, J.F., "Conceptual Structures: Information Processing in Mind and Machine", Addison Wesley, 1984.

[Steels85] Steels, L., "Second Generation Expert Systems", In: "Future Generation Computer Systems", North-Holland, 1985.

[Sterling87] Sterling, L., E.Y. Shapiro, "The Art of Prolog", The MIT Press, 1987.

[Stonebraker86] Stonebraker, M., L.A. Rowe, "The design of POSTGRES", In: "Proceedings of the ACM-SIGMOD Conference on Management of Data", 1986, pp. 340-355.

[Stonebraker88] Stonebraker, M. (et al.), "The POSTGRES Rule Manager", In: "IEEE Transactions on Software Engineering", Vol. 14, Nr. 7, Juli 1988, pp. 897-907.

[Stonebraker89] Stonebraker, M., M. Hearst, "Future Trends in Expert Database Systems", In: Kerschberg, L. (ed.), "Expert Database Systems, Proceedings from the Second International Conference", 1989, pp. 3-20.

[Sutcliffe88] Sutcliffe, A., "Jackson system development", 1988.

[Suwa82] Suwa, M. (et al.), "An Approach to Verifying Completeness and Consistency in a Rule-Based Expert System", Stanford University Report, STAN-CS-82-922, 1982.

[Suwa84] Suwa, M. (et al.), "Completeness and Consistency in Rule-Based Expert Systems", In: Buchanan, B.G. (ed.), "Rule-Based Expert Systems: The Mycin Experiments of the Stanford H ristic Programming Project", Addison-Wesley, 1984.

[Treur88] Treur, J., "Completeness and Definability in Diagnostic Expert Systems", Report P8805, University of Amsterdam, 1988.

[Turner87] Turner, W.S. (et al.), "System development methodology", 1987.

[VanHekken89] Hekken, M.C. van, P.J.M. van Oosterom, M.R. Woestenburg, "A Geographical Extension to the Relational Data Model", Internal Report FEL-89-B207, 1989.

[VanMelle84] Van Melle, W. (et al.), "EMYCIN: A Knowledge Engineer's Tool for constructing Rule-Based Expert Systems", In: Buchanan, B.G. (ed.), "Rule-Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project", Addison-Wesley, 1984.

[Vieille88] Vieille, L., "Recursive Query Processing: Fundamental Algorithms and the Ded Gin System", In: Cray P.M.D. & Lucas R.J. (eds.), "Prolog and Databases", Halsted Press/Wiley & Sons, 1988.

[Vogler88] Vogler, M., "KADS in Conventionele Context", In: "Proceedings AI Toepassingen '88, 1988, pp. 83-87.

[Walker88] Walker, R.F., "Prolexs: Een Object-georiënteerd Expertsysteem", In: "Proceedings AI Toepassingen", 1988.

[Weitzel89] Weitzel, J.R., L. Kerschberg, "Developing Knowledge-Based Systems: Reorganizing the System Development Life Cycle", In: "Communications of the ACM", Vol. 32, Nr. 4, 1989, pp. 482-488.

[Wensel88] Wensel, S., "The POSTGRES reference manual", Memo Nr. UCD/ERL M88/20, University of California, Berkeley, 1988.

[Wintraecken85] Wintraecken, J.J.V.R., "Informatie-Analyse volgens NIAM", Academic Service, 1985.

[Wintraecken88] Wintraecken, J.J.V.R., "Informatiemodellering volgens NIAM", In: "Proceedings Semantiek van Gegevensmodellen: Het Tijdperk na Codd", 1988, pp. 45-76.

[Wognum89] Wognum, P.M., Mars N.J.I., "Het Belang van Uitleg van Redeneringen", In: "Proceedings NAIC '89", Academic Service, 1989, pp. 189-198.

[Yourdon79] Yourdon, A., Constantine L.L., "Structured Design", Yourdon Press, 1979.

[Yu84] Yu, V.L., (et al.), "An Evaluation of MYCIN's Advice", In: B.G. Buchanan (ed.), "Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project", 1984, pp. 589-596

UNCLASSIFIED

REPORT DOCUMENTATION PAGE (MOD-NL)

1. DEFENSE REPORT NUMBER (MOD-NL) TD 89-5095	2. RECIPIENT'S ACCESSION NUMBER	3. PERFORMING ORGANIZATION REPORT NUMBER FEL-90-A012
4. PROJECT/TASK/WORK UNIT NO. 21896	5. CONTRACT NUMBER A89K602	6. REPORT DATE NOVEMBER 1990
7. NUMBER OF PAGES 91 (EXCL. RDP & DIST.LIST)	8. NUMBER OF REFERENCES 141	9. TYPE OF REPORT AND DATES COVERED FINAL REPORT

10. TITLE AND SUBTITLE
QUEST: QUALITY OF EXPERT SYSTEMS

11. AUTHOR(S)
JACQUES H.J. LENTING AND MICHAEL PERRE

12. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
TNO PHYSICS AND ELECTRONICS LABORATORY, P.O. BOX 96864,THE HAGUE, THE NETHERLANDS
UNIVERSITY OF LIMBURG, P.O. BOX 616, MAASTRICHT, THE NETHERLANDS
RESEARCH INSTITUTE FOR KNOWLEDGE SYSTEMS, P.O. BOX 463, MAASTRICHT, THE NETHERLANDS

13. SPONSORING/MONITORING AGENCY NAME(S)
MOD-NL DIRECTOR DEFENSE RESEARCH AND DEVELOPMENT

14. SUPPLEMENTARY NOTES
THE PHYSICS AND ELECTRONICS LABORATORY IS PART OF THE NETHERLANDS ORGANIZATION FOR APPLIED SCIENTIFIC RESEARCH

15. ABSTRACT (MAXIMUM 200 WORDS, 1044 POSITIONS)

THIS REPORT CONTAINS A SUMMARY OF THE RESULTS OF THE TECHNOLOGY PROJECT 'QUEST: QUALITY OF EXPERT SYSTEMS', CARRIED OUT UNDER COMMISSION OF THE DUTCH MINISTRY OF DEFENCE, DIRECTOR DEFENSE RESEARCH AND DEVELOPMENT. PARTICIPANTS IN THE PROJECT ARE TNO PHYSICS AND ELECTRONICS LABORATORY (FEL-TNO), UNIVERSITY OF LIMBURG (RL) AND THE RESEARCH INSTITUTE FOR KNOWLEDGE SYSTEMS (RIKS). AFTER AN ANALYSIS OF THE PROBLEMS ENCOUNTERED IN EXPERT SYSTEMS DEVELOPMENT, A QUALITY FRAMEWORK IS DEVELOPED WHICH VIEWS THE QUALITY PROBLEM FROM THREE PERSPECTIVES: THE QUALITY OF THE DEVELOPMENT PROCESS, THE QUALITY OF THE SPECIFICATIONS AND THE QUALITY OF THE EXPERT SYSTEM AS A PRODUCT. IN ORDER TO GET A BETTER GRASP OF THE PROBLEM A NUMBER OF METHODS AND TECHNIQUES, DERIVED FROM CONVENTIONAL AND ARTIFICIAL INTELLIGENCE SYSTEMS DEVELOPMENT, ARE REVIEWED. SECONDLY THE CONCEPTUAL SIMILARITIES BETWEEN DATABASES AND KNOWLEDGEBASES ARE STRESSED. THE USE OF CONVENTIONAL SPECIFICATION METHODS, IN PARTICULAR NIJSSENS INFORMATION ANALYSIS METHODOLOGY (NIAM), IS CONSIDERED. IN ADDITION TO THIS ALGORITHMS FOR PRESERVING CONSISTENCY AND INTEGRITY OF THE KNOWLEDGEBASE ARE COMPARED. THIRDLY THE MODULARITY AND STRUCTURE OF KNOWLEDGEBASES IS EXAMINED, TOGETHER WITH THE APPLICABILITY OF CONVENTIONAL TESTING METHODOLOGIES IN EXPERT SYSTEMS. LASTLY IT IS DEMONSTRATED THAT THE INTEGRATION OF DATABASE THEORY AND ARTIFICIAL INTELLIGENCE SIGNIFIES A STEP IN THE DIRECTION OF A BETTER QUALITY CONTROL OF EXPERT SYSTEMS.

16. DESCRIPTORS	DESCRIPTORS
ARTIFICIAL INTELLIGENCE	DATABASES
EXPERT SYSTEMS	IDENTIFIERS
QUALITY CONTROL	KNOWLEDGEBASES

17a. SECURITY CLASSIFICATION (OF REPORT) UNCLASSIFIED	17b. SECURITY CLASSIFICATION (OF PAGE) UNCLASSIFIED	17c. SECURITY CLASSIFICATION (OF ABSTRACT) UNCLASSIFIED
18. DISTRIBUTION/AVAILABILITY STATEMENT UNLIMITED AVAILABLE		17d. SECURITY CLASSIFICATION (OF TITLES) UNCLASSIFIED

UNCLASSIFIED